(2)

An Annual Report
Grant No. N00014-91-J-1102

October 1, 1990 - September 30, 1991

# THE STARLITE PROJECT

# PROTOTYPING REAL-TIME SOFTWARE

Submitted to:

Scientific Officer Code: 1211
Dr. James G. Smith
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217-5000

Submitted by:

R. P. Cook
Associate Professor

S. H. Son
Assistant Professor

SEAS Report No. UVA/525449/CS92/101
October 1991

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF
ENGINEERING
& APPLIED SCIENCE

University of Virginia
Thornton Hall
Charlottesville, VA 22903

An Annual Report
Grant No. N00014-91-J-1102

October 1, 1990 - September 30, 1991

# THE STARLITE PROJECT

# PROTOTYPING REAL-TIME SOFTWARE

Submitted to:

Scientific Officer Code: 1211
Dr. James G. Smith
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217-5000

Submitted by:

R. P. Cook
Associate Professor

S. H. Son
Assistant Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/525449/CS92/101
October 1991

Copy No. 7

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

| | | |
|---|---|---|
| 1. AGENCY USE ONLY (*Leave blank*) | 2. REPORT DATE<br>October 1991 | 3. REPORT TYPE AND DATES COVERED<br>Annual 10/1/90 - 9/30/91 |

**4. TITLE AND SUBTITLE**
The Starlite Project - Prototyping Real-Time Software

**5. FUNDING NUMBERS**
N00014-91-J-1102

**6. AUTHOR(S)**
R. P. Cook
S. H. Son

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
University of Virginia
School of Engineering and Applied Science
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903-2442

**8. PERFORMING ORGANIZATION REPORT NUMBER**
UVA/525449/CS92/101

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5000

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
Unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** (*Maximum 200 words*)

NA - Letter Report

**14. SUBJECT TERMS**
real time systems, prototyping, distributive databases, operating systems, scheduling

**15. NUMBER OF PAGES**
64(5 plus appendix)

**16. PRICE CODE**

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT<br>Unclassified | 18. SECURITY CLASSIFICATION OF THIS PAGE<br>Unclassified | 19. SECURITY CLASSIFICATION OF ABSTRACT<br>Unclassified | 20. LIMITATION OF ABSTRACT<br>UL |

# Annual Letter Report for FY91

## Contract Information

The StarLite Project - Prototyping Real-Time Software
N00014-91-J-1102

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2205
son@cs.virginia.edu

10/1/90 - 9/30/91
ONR Program Manager: James G. Smith

## Technical Objectives and Issues

The goals of StarLite project are twofold. First, we investigate new technologies associated with the construction of real-time, distributed operating systems, database systems, and communication networks. Secondly, the research will produce tools and experimental systems, using the StarLite software prototyping environment. The issues being addressed are priority-based scheduling, methodology for developing prototyping tools, system integration, operating system support and interface issues, and fault-tolerance in real-time computing.

## Approach

Our research is directed towards discovering a set of design principles and developing efficient algorithms for distributed real-time operating systems and database systems. In addition to theoretical study, we are also developing experimental systems and prototyping tools for performance evaluation of new technology.

## Accomplishments and Significance

One of the most important achievements in FY91 is the development of new scheduling algorithms based on the idea of adjusting the serialization order of active transactions dynamically. This is the first successful attempt to integrate benefits of pessimistic and optimistic approaches for transaction scheduling. When compared with conventional transaction scheduling algorithms (e.g., two-phase locking), our algorithms significantly improve the percentage of high priority transactions that meet the deadline. We also have developed priority-ordered deadlock avoidance algorithms for real-time resource management, and replication control algorithms for distributed real-time databases. These algorithms will be very efficient for distributed real-time systems, in which replicated resources should be managed to support consistency, while satisfying timing constraints. Using StarLite prototyping environment, we have implemented those algorithms and demonstrated that they provide higher level of concurrency and greater flexibility in meeting timing requirements.

We have developed a new paradigm for multiprocessor real-time systems, and implemented a parallel programming interface based on our paradigm. Our new paradigm has created new research opportunities for operating systems and databases for parallel computing systems with timing constraints. For example, using the new programming interface, we have developed PRDB, an experimental real-time database system that runs on an emulated tightly-coupled, shared-memory multiprocessor system in the StarLite environment. It provides a general paradigm for exlpoiting parallelism and different real-time scheduling policies. This experimental system has been used for investigating implementation techniques for parallel database systems and the impact of multiprocessor technology on operating systems design.

We have developed a suite of database systems on several platforms, such as StarLite, ARTS, and UNIX, and utilized them as system integration testbeds. S' .ce a real-time system must operate in the context of operating system services, correct functioning and timing behavior of the system depends heavily on the operating system interfaces. We have developed a multi-thread database server, called RTDB, for ARTS real-time operating system kernel. The RTDB now supports application programmatic interface and graphic user interface. Our experimental systems achieve other goal of this project—to transfer technology developed under the StarLite project to Navy, DoD, and other research organizations. Currently, Naval Ocean Systems Center in San Diego, California, is using RTDB for their distributed real-time experiments.

## Plans for FY92

We will continue to expand the module hierarchy of the StarLite environment by including modules for optimistic scheduling and system recovery, and graphic user interface. We will also investigate system support requirements for real-time applications, and then evaluate the StarLite from that perspective. In addition, we will refine the current single processor and multiprocessor implementation. Furthermore, we will begin work on a distributed version StarLite operating system that will be integrated with our distributed database kernel.

We will investigate scheduling and concurrency control algorithms, and perform experiments for their evaluation. We will also implement replication control algorithms and recovery algorithms in the StarLite for their performance evaluation. With respect to experimental database systems, we will add real-time transaction features and evaluate them for realistic applications. Our findings and new technology developed in this project will continue to be transfered to other Navy organizations.

## Presentations, Publications, and Honors

### • Presentations

- Cook, The StarLite Project, University of North Carolina at Charlotte.
- Cook, The StarLite Project, University of Hawaii.
- Cook, The StarLite Project, Microsoft Corporation.
- Son, Real-Time Database Systems, NOSC Code 413 DC$^2$ Quarterly Review.
- Son, Real-Time Database Systems, Sogang University.
- Son, Prototyping Approach to Database Research, Electronics and Telecommunications Research Institute.
- Son, Scheduling Real-Time Transactions, Seoul National University.

### • Book Chapters

(1) Cook, R. P., "The StarLite Operating System," *Operating Systems for Mission-Critical Computing*, K. Gordon, P. Hwang, and A. Agrawala (Editors), ACM Press, 1991.

(2) Cook, R. P., "Modula-2," *Encyclopedia of Computer Science*, (to appear).

(3) R. Cook, L. Hsu, and S. H. Son, "Real-Time, Priority-Ordered, Deadlock Avoidance Algorithms," in *Foundations of Real-Time Computing: Scheduling and Resource Management*, A. Van Tilborg

and G. M. Koob (Editors), Kluwer Academic Publishers, 1991.

(4) S. H. Son, "On Priority-based Synchronization Protocols for Distributed Real-Time Databa" se Systems," in *Distributed Databases in Real-Time Control*, E. Knuth and M. Rodd (Editors), Pergamon Press, 1990.

(5) S. H. Son, Y. Lin, and R. Cook, "Concurrency Control in Real-Time Database Systems," in *Foundations of Real-Time Computing: Scheduling and Resource Management*, A. Van Tilborg and G. M. Koob (Editors), Kluwer Academic Publishers, 1991.

(6) S. H. Son, R. Cook, J. Lee, and H. Oh, "New Paradigms for Real-Time Database Systems," in *Real-Time Programming*, K. Ramamritham and W. Halang (Editors), Pergamon Press, (to appear).

(7) S. H. Son and S. Park, "Scheduling Transactions for Distributed Time-Critical Applications," in *Advances in Distributed Systems*, T. Casavant and M. Singhal (Editors), IEEE Computer Society, (to appear).

• **Refereed Publications**

(1) S. H. Son, "Reconstruction of Distributed Databases," *Computer Systems Science and Engineering*, vol. 5, no. 4, October 1990.

(2) P. Shebalin, S. H. Son, and C. Chang, "An Approach to Software Safety Analysis in Distributed Systems," *Computer Systems Science and Engineering*, vol. 6, no. 2, April 1991.

(3) L. Sha, R. Rajkumar, S. H. Son, and C. Chang, "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, vol. 40, no. 7, July 1991.

(4) S. H. Son, "An Environment for Integrated Development and Evaluation of Real-Time Distri" buted Database Systems," *Journal of Systems Integration*, (to appear).

(5) S. H. Son, J. Ratner, S. Chiang, "StarBase: A Simulation Laboratory for Distributed Database Research," *Journal of Computer Simulation*, (to appear).

(6) Cook, R. P. and H. Oh, "The StarLite Project," *Frontiers 90 Conference on Massively Parallel Computation*, October 1990.

(7) Y. Lin and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serializ" ation Order," *11th IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.

(8) S. H. Son, C. Iannacone, and R. Beckinger, "Integrating Databases with Real-Time Computing Systems," *IEEE Southeastcon '91*, Williamsburg, Virginia, April 1991.

(9) S. H. Son, M. Poris, and C. Iannacone, "Implementing a Distributed Real-Time Database Manager," *The Second International Symposium on Database Systems for Advanced Applications (DASFAA '91)*, Tokyo, Japan, April 1991.

(10) H. Kang and S. H. Son, "A Hierarchical Export/Import Scheme for Data Sharing in a Federated Distribu" ted Database System," *The Second International Symposium on Database Systems for Adv[illegible]d Applications (DASFAA '91)*, Tokyo, Japan, April 1991.

(11) S. H. Son, P. Wagle, and S. Park, "Real-Time Database Scheduling: Design, Implementation, and Performance Evalu'' ation," *The Second International Symposium on Database Systems for Advanced Applications (DASFAA '91),* Tokyo, Japan, April 1991.

(12) R. P. Cook, S. H. Son, H. Y. Oh, and J. Lee, "New Paradigms for Real-Time Database Systems," *8th IEEE Workshop on Real-Time Operating Systems and Software,* Atlanta, Georgia, May 1991.

(13) S. H. Son, C. Iannacone, and M. Poris, "RTDB: A Real-Time Database Manager for Time-Critical Applications," *Euromicro Workshop on Real-Time Systems,* Paris, France, June 1991.

(14) S. H. Son and S. Chiang, "Concurrent Checkpointing Algorithms for Distributed Database Systems," *Fourth International Conference on Parallel and Distributed Computing and Systems,* Washington, DC, October 1991 (to appear).

(15) S. H. Son, S. Park, and Y. Lin, "An Integrated Real-Time Locking Protocol," *Eighth IEEE International Conference on Data Engineering,* Phoenix, Arizona, February 1992 (to appear).

**• Honors and Recognition**

- Cook, Program Committee, Eighth IEEE Workshop on Real-Time Software and Operating Systems, Atlanta, Geotgia (1991).
- Son, ACM Distinguished Lecturer, 1991 — 1993.
- Son, Chair, Technical Activities Committee, Korean Computer Scientists and Engineers Association.
- Son, Program Committee, IEEE Symposium on Reliable Distributed Systems, Huntsville, Alabama (1990).
- Son, Program Committee, ACM SIGMOD International Conference on Management of Data, Denver, Colorado (1991).
- Son, Program Committee, IEEE Workshop on Real-Time Operating Systems and Software, (1992).
- Son, Program Committee, International Workshop on Transaction and Query Processing, (1992).
- Son, Program Committee, International Symposium on Database Systems for Advanced Applications, (1993).
- Son, Panelist, International Conference on Very Large Data Bases (VLDB '91), on the panel "Real-Time Databases," (1991).
- Son, Session Chair, Ninth IEEE Symposium on Reliable Distributed Systems, (1990).
- Son, Session Chair, 12th IEEE Real-Time Systems Symposium, (1991).
- Son, Invited Paper, "A Prototyping Approach to Distributed Database Research," *Database Review,* vol. 6, October 1990.
- Son, Invited Paper, "Real-Time Database Systems: A New Challenge," *Data Engineering,* Special Issue on Directions for Future Database Research and Development, vol. 13, no. 4, December 1990.

- Son, Invited Paper, ''Techniques for Database Recovery in Distributed Environments,'' *Management and Organization of Automation*, Kluwer Bedrijfswetenschappen, January 1991.

**• Students and Post-Docs**

Seog Park, (Post-Doc), real-time transaction scheduling

Marc Poris (Research Associate), database integration with real-time kernel

Shi-Chin Chiang (Ph.D. student), checkpointing in distributed systems

Lee Hsu (Ph.D student), priority-based resource management

Yingfeng Oh (Ph.D. student), real-time multiprocessor operating systems

Juhnyoung Lee (Ph.D. student), schedulers for real-time databases

Young-Kuk Kim (Ph.D. student), interface issues for real-time databases

Ambar Sarkar (Ph.D. student), real-time fault-tolerant network protocols

Prasad Wagle (M.S. student), dynamic priority scheduling

Yi Lin (M.S. student), priority-based contention protocols

Carmen Iannacone (M.S. student), multi-thread real-time database server

Robert Beckinger (M.S. student), support for temporal information

Sprios Kouloumbis (M.S. student), replication control

Savita Shamsunder (M.S. student), optimistic concurrency control protocols

Stavros Yannopolous (M.S. student), experimental database manager

Fengjie Zhang (M.S. student), majority consensus for real-time databases

David Bake (M.S. student), distributed real-time transaction processing

Richard McDaniel(B.S. student), prototyping environment

**• Number and Types of Degrees Granted**

M.S. degrees granted to 6 students.

# APPENDIX

# Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order

Yi Lin and Sang H. Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

## ABSTRACT

Time-critical scheduling in real-time database systems has two components: real-time transaction scheduling, which is related to task scheduling in real-time operating systems, and concurrency control, which can be considered as operation level scheduling. Most current research in this area only focuses on the transaction scheduling aspect while the concurrency control part remains untouched. In the paper, a new concurrency control algorithm for real-time database systems is proposed, by which real-time scheduling and concurrency control can be integrated. The algorithm is based on a priority-based locking mechanism to support time-critical scheduling by adjusting the serialization order dynamically in favor of high priority transactions. Furthermore, it does not assume any knowledge about the data requirements or execution time of each transaction. This makes the algorithm very practical.

## 1. Introduction

Compared with traditional databases, the ability to meet the deadlines of transactions is vital to a real-time database. In other words, the timeliness of results can be as important as their correctness in real-time database systems. Deadlines constitute the timing constraints of transactions. A deadline is said to be *hard* if it cannot be missed or else the result is useless. If a deadline can be missed, it is a *soft* deadline. With soft deadlines, the usefulness of a result may decrease after the deadline is missed. Another important characteristic of real-time transactions is criticality, which represents the importance of a transaction. Deadline and criticality are two orthogonal factors that play important roles in real-time database systems [Son88].

This work was supported in part by ONR under contract # N00014-88-K-0245, by DOE under contract # DE-FG05-88-ER25063, by IBM FSD, and by CIT under contract # CIT-INF-90-011.

The goal of scheduling in real-time database systems is twofold: to meet timing constraints and to enforce data consistency. In real-time operating systems, scheduling is usually at the task level subject only to timing constraints. Data consistency constraints are not involved. In conventional database systems, meeting the deadline is typically not addressed. Scheduling is at operation level in that the basic unit of a schedule is operation. The only goal of such scheduling is to guarantee data consistency. Scheduling in the real-time database systems is a combination of the two scheduling mechanisms [Son90c].

Real-time task scheduling methods can be extended for real-time transaction scheduling while concurrency control protocols are still needed for operation scheduling to maintain data consistency. However, the integration of the two mechanisms in real-time database systems is not trivial. The general approach is to utilize existing concurrency control protocols, especially 2PL, and to apply time-critical transaction scheduling methods that favor more urgent transactions [Abb88, Sha90, Son89b]. Such approaches have the inherent disadvantage of being limited by the concurrency control method upon which they are based, since all existing concurrency control methods synchronize concurrent data access of transactions by the combination of two measures: blocking and roll-backs of transactions. Both are barriers to time-critical scheduling. The conservative two-phase locking (2PL) protocol [Bern87] and the optimistic methods [Bok87, Kung81] are examples of the two extremes. In real-time database systems, blocking may cause priority inversion when a high priority transaction is blocked by lower priority transactions [Sha88]. The alternative is to abort the low priority transactions if they block a high priority transaction. This wastes the work done by the aborted transactions and in turn also has a negative effect on time-critical scheduling.

Concurrency control protocols induce a serialization order among conflicting transactions. In non-real-time concurrency control protocols, timing constraints are not a factor in the construction of this order. This is obviously a drawback for real-time database systems.

For example, with the 2PL method, the serialization order is dynamically constructed and corresponds to the order in which the conflicting transactions access the shared data objects. In other words, the serialization order is bound to the past execution history with no flexibility. When a transaction $T_H$ with a higher priority requests an exclusive lock which is being held by another transaction, $T_L$, with a lower priority, the only choices are either aborting $T_L$ or letting $T_H$ wait for $T_L$. Neither choice is satisfactory and thus the performance is degraded.

Based on the argument that timing constraints may be more important than data consistency in real-time database systems, attempts have been made to satisfy the timing constraints by sacrificing database consistency temporarily to some degree [Kor90, Lin89, Vrb88]. It is based on a new consistency model of real-time databases, in which maintaining *external data consistency* has priority over maintaining *internal data consistency*. Although in some applications weaker consistency is acceptable [Gar83], a general-purpose consistency criterion that is less stringent than serializability has not yet been proposed. The problem is that temporary inconsistencies may affect active transactions and so the commitment of these transactions may still need to be delayed until the inconsistencies are removed; otherwise even committed transactions may need to be rolled back. However, in real-time systems, some actions are not reversible. In addition, incorrect data may spread within the database. This makes inconsistency removal a very difficult task. Before any breakthrough is made in this direction, serializability seems to be the only correctness criterion for us to live with.

Satisfying the timing constraints while preserving data consistency requires the concurrency control algorithms to accommodate timeliness of transactions as well as to maintain data consistency. This is the very goal of our work. In real-time database systems, timeliness of a transaction is usually combined with its criticality to take the form of the priority of that transaction. Various ways of assigning priority and their effects have been discussed in [Stan88, Hua89, Buch89].

For a concurrency control algorithm to accommodate the timeliness of transactions, the serialization order it produces should reflect the priority of transactions [Son90b]. However, this is often hindered by the past execution history of transactions. For example, a higher priority transaction may have no way to precede a lower priority transaction in the serialization order due to previous conflicts. The result is that either the lower priority transaction has to be aborted or the high priority transaction suffers blocking. If the information about data requirements and execution time of each transaction is available beforehand, off-line preanalysis can be performed to avoid conflicts [Sha90]. This is exactly what is done in many real-time task scheduling protocols. However, such approach may have to delay the starting of some transactions, even if they have high priorities, and may reduce the concurrency level in the system. This, in return, may lead to the violation of the timing constraints and degrade the system performance [Son90].

What we need is a concurrency control algorithm that allows transactions to meet the timing constraints as much as possible without reducing the concurrency level of the system in the absence of any *a priori* information [Son89]. The algorithm presented in this paper features such ability. It has the flavor of both locking and optimistic methods. Transactions write into the database only after they are committed. By using a priority-dependent locking protocol, the serialization order of active transactions is adjusted dynamically, making it possible for transactions with higher priorities to be executed first so that higher priority transactions are never blocked by uncommitted lower priority transactions, while lower priority transactions may not have to be aborted even in face of conflicting operations. The adjustment of the serialization order can be viewed as a mechanism to support time-critical scheduling. For example, $T_1$ and $T_2$ are two transactions with $T_1$ having a higher priority. $T_2$ writes a data object $x$ before $T_1$ reads it. In 2PL, even in the absence of any other conflicting operations between these two transactions, $T_1$ has to either abort $T_2$ or be blocked until $T_2$ releases the write lock. That is because the serialization order $T_2 \rightarrow T_1$ is already determined by the past execution history. $T_1$ can never precede $T_2$ in the serialization order. In our method, when such conflict occurs, the serialization order of the two transactions will be adjusted in favor of $T_1$, i.e. $T_1 \rightarrow T_2$, and neither is $T_1$ blocked nor is $T_2$ aborted. In addition, the locking protocol in the algorithm is free from deadlocks.

The remainder of this paper is organized as follows. The details of the proposed algorithm are described in the next section. The transaction scheduling aspect of the algorithm is discussed in Section 3. Section 4 presents an informal argument on the correctness of the method. Section 5 presents an example to show how the algorithm works. Finally concluding remarks appear in Section 6.

## 2. The Proposed Algorithm

The environment we assume for the implementation is a single processor with randomly arriving transactions. Each transaction is assigned an *initial priority* and a *start-timestamp* when it is submitted to the system. The initial priority can be based on the deadline and the criticality of the transaction. The start-timestamp is appended to the initial priority to form the *actual priority* that is used in scheduling. When we refer to the priority of a transaction, we always mean the actual priority with the start-timestamp appended. Since the start-timestamp is unique, so is the priority of each transaction. The priority of transactions with the same initial priority is distinguished by their start-timestamps.

All transactions that can be scheduled are place in a ready queue, *R_Q*. Only transactions in *R_Q* are scheduled for execution. When a transaction is *blocked*, it is removed from *R_Q*. When a transaction is *unblocked*, it is inserted into *R_Q* again, but may still be waiting to be assigned the CPU. A transaction is said to be *suspended* when it is not executing, but still in *R_Q*. When a transaction is doing I/O operation, it is blocked. Once it completes, it is usually unblocked. We will discuss the CPU scheduling policy in the next section.

The execution of each transaction is divided into three phases: the read phase, the wait phase and the write phase. This is similar to the optimistic methods. During the read phase, a transaction reads from the database and writes to its local workspace. After it completes, it waits for its chance to commit in the wait phase. If it is committed, it switches into the write phase during which all its updates are made permanent in the database. A transaction in any of the three phases is called *active*. If an active transaction is in the write phase, then it is committed and writing into the database. The proposed algorithm takes an approach of integrated schedulers in that it uses 2PL for read-write conflicts and the Thomas' Write Rule (TWR) for write-write conflicts. The following is the outline of a transaction:

```
transaction = { tbegin();
                read phase;
                twait();
                twrite();
              } .
```

All the procedures will be defined later in this section.

In our algorithm, there are various data structures that need to be read and updated in a consistent manner. Therefore we use critical sections of various classes to group the various data structures to allow maximum concurrency. We also assume that each assignment statement of global data is executed atomically. The following are some useful notations:

| | |
|---|---|
| *id*: | id of this transaction |
| *read_trset*: | set of ids of transactions in the read phase |
| *wait_trset*: | set of ids of transactions in the wait phase |
| *write_trset*: | set of ids of transactions in the write phase |
| *tscnt*: | final-timestamp count of the system |
| *ts*: | final-timestamp of this transaction |
| *ts(T)*: | final-timestamp value of transaction T |
| *priority(T)*: | priority value of transaction T |
| $r_i[x]$: | transaction $i$ reads data object $x$. |
| $w_i[x]$: | transaction $i$ writes data object $x$. |
| $pw_i[x]$: | transaction $i$ prewrites data object $x$. |
| *rlock(T,x)*: | transaction T holds a read lock on data object $x$ |
| *wlock(T,x)*: | transaction T holds a write lock on data object $x$ |
| $<_i$: | begin critical section of class $i$ |
| $_i>$: | end critical section of class $i$ |

## 2.1. Read Phase

The read phase is the normal execution of the transaction except that write operations are performed on the private data copies in the local workspace of the transaction instead of on the data objects in the database. We call such write operations *prewrite*. One advantage of this prewrite operation is that when a transaction is aborted, all that has to be done for recovery is to simply discard the data in its local workspace. No rollback is needed because no changes have been made in the database.

The read-prewrite or prewrite-read conflicts between active transactions are synchronized during this phase by a priority-based locking protocol. Before a transaction can perform a read (resp. prewrite) operation on a data object, it must obtain the read (resp. write) lock on that data object first. If a transaction reads a data object that has been written by itself, it gets the private copy in its own workspace immediately and no read lock is needed. In the rest of the paper, when we refer to read operations, we exclude such read operations because they do not incur any dependency among transactions.

Each lock contains the priority of the transaction holding the lock as well as other usual information such as the lock holder id and the lock type, etc. The locking protocol is based on the principle that higher priority transactions should complete before lower priority transactions. That means if two transactions conflict, the higher priority transaction should precede the lower priority transaction in the serialization order. With our CPU scheduling policy, which will be described in the next section, a high priority transaction is scheduled to commit before a low priority transaction most of the time. If a low priority transaction does complete before a high priority transaction, it is required to wait until it is sure that its commitment will not cause the abortion of a higher priority transaction. Since transactions do not write into the database during the read phase, write-write conflicts need not be considered here.

Suppose active transaction $T_1$ has higher priority than active transaction $T_2$. We have the following four possibilities of conflict and the transaction dependencies they set in the serialization order:

(1) $r_{T_1}[x]$, $pw_{T_2}[x]$ => $T_1 \rightarrow T_2$

(2) $pw_{T_1}[x]$, $r_{T_2}[x]$ => $T_1 \rightarrow T_2$ (delayed reading) or $T_2 \rightarrow T_1$ (immediate reading)

(3) $r_{T_2}[x]$, $pw_{T_1}[x]$ => $T_2 \rightarrow T_1$

(4) $pw_{T_2}[x]$, $r_{T_1}[x]$ => $T_1 \rightarrow T_2$ (immediate reading) or $T_2 \rightarrow T_1$ (delayed reading)

Case (1) meets the principle of completing high priority transactions before low priority ones. In case (2), following our principle, we should choose delayed reading, i.e. $T_2$ should not read $x$ until $T_1$ has committed and written $x$ in the database. Case (3) violates our principle. In this case, unless it is already committed, $T_2$ is usually aborted because otherwise $T_2$ must commit before $T_1$ and thus will block $T_1$. However, if $T_2$ has already finished its work, i.e. in the wait phase, we should avoid aborting it because aborting a transaction which has completed its work imposes a considerable penalty on the the system performance. In the meantime, we still do not want $T_1$ to be blocked by $T_2$. Therefore when such conflict occurs and $T_2$ is in the wait phase, we do not abort $T_2$ until $T_1$ is committed, hoping that $T_2$ may get a chance to commit before $T_1$ commits. In case (4), if $T_2$ is already committed and in the write phase, we should delay $T_1$ so that it reads $x$ after $T_2$ writes it. This blocking is not a serious problem for $T_1$ because $T_2$ is already in the write phase and is expected to finish writing $x$ soon. $T_1$ can read $x$ as soon as $T_2$ finishes writing $x$ in the database, not necessarily after $T_2$ completes the whole write phase. Therefore $T_1$ will not be blocked for a long time. Otherwise, if $T_2$ is not committed yet, i.e. either in the read phase or in the wait phase, $T_1$ should read $x$ immediately because that is in accordance with the principle.

As transactions are being executed and conflicting operations occur, all the information about the induced dependencies in the serialization order needs to be retained. To do this, we associate with each transaction two sets, *before_trset* and *after_trset*, and a count, *before_cnt*. The set *before_trset* (resp. *after_trset*) contains all the active lower priority transactions that must precede (resp. follow) this transaction in the serialization order. *before_cnt* is the number of the higher priority transactions that precede this transaction in the serialization order. When a conflict occurs between two transactions, their dependency is set and their values of *before_trset*, *after_trset*, and *before_cnt* will be changed correspondingly.

By summarizing what we discussed above, we define the locking protocol as follows:

LP1. Transaction $T$ requests a read lock on data object $x$.

```
<0<1 for $t \in \{T_i \mid wlock(T_i,x) \wedge Ti \neq T\}$ do
        if (priority (t) > priority (T)
                $\vee\ t \in$ write_trset)   /* Case 2, 4 */
        then deny the lock and exit;
        endif
     enddo
     for $t \in \{T_i \mid wlock(T_i,x) \wedge Ti \neq T\}$ do
                 /* Case 4 */
      if ($t \in$ before_trset$_T$)  then abort t;
      else if ($t \notin$ after_trset$_T$)
            then
              after_trset$_T$ := after_trset$_T \cup \{t\}$;
              before_cnt$_t$ := before_cnt$_t$ + 1;
            endif
      endif
     enddo
     grant the lock;
 1> 0>
```

LP2. Transaction $T$ requests a write lock on data object $x$.

```
<0<2<3 for $t \in \{T_i \mid rlock(T_i,x) \wedge T_i \neq T\}$ do
          if (priority (t) > priority (T) )
          then /* Case 1 */
            if ($T \notin$ after_trset$_t$)
            then
              after_trset$_t$ := after_trset$_t \cup \{T\}$;
              before_cnt$_T$ := before_cnt$_T$ + 1;
            endif
    2> 3>
        else
          if ($t \in$ wait_trset)  /* Case 3 */
          then
            if ($t \in$ after_trset$_T$) then abort t;
            else
             before_trset$_T$ := before_tr[illegible]
            endif
          else if ($t \in$ read_trset)
               then abort t;
               endif
          endif
        endif
       grant the lock;
 0>
```

The critical sections of class 0 guarantee that lock requests are processed sequentially, probabl_ by a lock manager. LP1 and LP2 are actually two procedures of the lock manager that are executed when a lock is requested. When a lock is denied due to a conflicting lock, the request is suspended until that conflicting lock is released. Then the locking protocol is invoked once again from the very beginning to decided whether the lock can be granted now. Fig. 1 shows the lock compatibility tables in which the compatiblities are expressed by possible actions taken when conflicts occur. The compatibility depends on the priorities of the transactions holding and requesting the lock and the phase of the lock holder as well as the lock types. Even with the same lock types, different actions may be taken, depending on the priorities of the lock holder and the lock requester. Therefore a table entry may have more than one blocks reflecting the different possible actions.

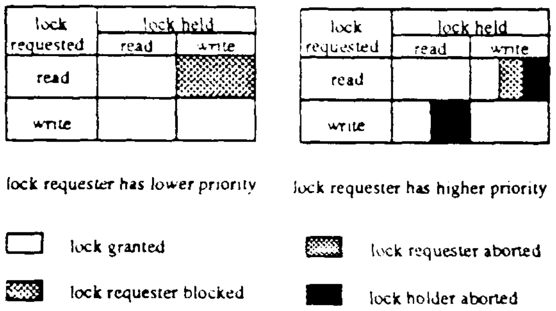

lock requested
lock held
read
write
read
write
lock requester has lower priority
lock requested
lock held
read
write
read
write
lock requester has higher priority
lock granted
lock requester blocked
lock requester aborted
lock holder aborted

Fig. 1 Lock Compatibility Table

Note that a data object may be both read locked and write locked by several transactions simultaneously with our locking protocol. Unlike 2PL, locks are not classified simply as shared locks and exclusive locks. Fig. 2 summarizes the lock compatibility of 2PL with the *High Priority* scheme in which high priority transactions never block for a lock held by a low priority transaction [Abb88]. By comparing Fig. 1 with Fig. 2, it is obvious that our locking protocol is much more flexible, thus incurs less blocking and abort. Note that in Fig. 1, the abort of lower priority transactions in the wait phase is also included. In our locking protocol, a high priority transaction is never blocked or aborted due to conflict with an uncommitted lower priority transaction. The probability of aborting a lower priority transaction should be less than that in 2PL under the same conditions. An analytical model may be used to estimate the exact probability, but that is beyond the scope of this paper.

Transactions are released for execution as soon as they arrive. The following procedure is executed when a transaction is started:

```
tbegin = (
  before_trset := $\emptyset$;
  after_trset := $\emptyset$;
  before_cnt := 0;
  read_trset := read_trset $\cup$ {id};
  R_Q := R_Q $\cup$ {id};
 ).
```

Then the transaction is in the read phase. When it tries to read or prewrite a data object, it requests the lock. The lock may be granted or not according to the locking protocol. Transactions may be aborted when lock requests are processed. To abort a transaction, the following procedure is called:

```
tabort = (
  release all locks;
 $<_2$for $t \in$ after_trset do
      before_cnt$_t$ := before_cnt$_t - 1$;
      if (before_cnt$_t = 0 \wedge t \in$ wait_trset)
    then unblock t;
      endif
   enddo
 $_2>$
   if (id $\in$ read_trset)
   then read_trset := read_trset - {id};
   else if (id $\in$ write_trset)
       then write_trset := write_trset - {id};
       else if (id $\in$ wait_trset)
            then wait_trset := wait_trset - {id};
            endif
       endif
   endif
 ).
```

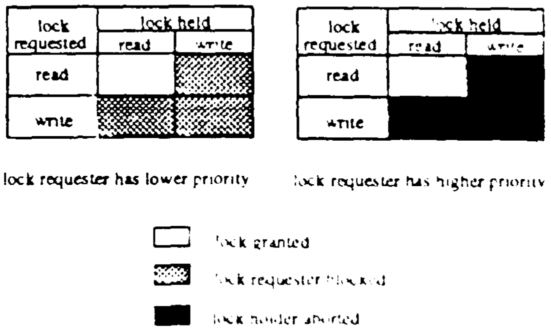

lock requested
lock held
read
write
read
write
lock requester has lower priority
lock requested
lock held
read
write
read
write
lock requester has higher priority
lock granted
lock requester blocked
lock holder aborted

Fig. 2 Lock Compatibility Table of 2PL

The critical section of class 2 in the procedure also appears in LP2. This ensures the mutual exclusion on *after_trset*. To be precise, mutual exclusion is only needed between LP2 and the procedure. Transactions can be in the critical section of the procedure simultaneously, because each transaction in the procedure *tabort* only access its own *after_trset*.

## 2.2. Wait Phase

The wait phase allows a transaction to wait until it can commit. A transaction $T$ can commit only if all transactions with higher priorities that must precede it in the serialization order are either committed or aborted. Since *before_cnt* is the number of such transactions, $T$ can commit only if its *before_cnt* becomes zero. A transaction in the wait phase may be aborted due to two reasons. The first one is that since $T$ is not committed yet and still holding all the locks, by the locking protocol it may be aborted due to a conflicting lock request by a higher priority transaction. The second reason is the commitment of a higher priority transaction that must follow $T$ in the serialization order. When such a transaction commits, it finds $T$ in *before_trset* and aborts $T$. Once a transaction in the wait phase gets its chance to commit, i.e. its *before_cnt* goes to zero, it switches into the write phase and release all its read locks. A final-timestamp is assigned to it, which is the absolute serialization order. The procedure is as follows:

```
twait = (
    wait_trset := wait_trset ∪ {id};
    read_trset := read_trset - {id};
    waiting := TRUE;
    while(waiting) do
      <_1 if (before_cnt = 0)      /* if can commit */
          then  /* switching into write phase */
                wait_trset := wait_trset - {id};
                write_trset := write_trset ∪ {id};
                ts := tscnt;
                tscnt := tscnt + 1;
                for t ∈ before_trset do
                  if (t ∈ read_trset ∨ t ∈ wait_trset )
                  then abort t;
                  endif
                enddo
      _1>
                waiting := FALSE
          else block;
          endif
    enddo
    release all read locks;
<_3 for t ∈ after_trset do
          if (t ∈ read_trset ∨ t ∈ wait_trset )
          then before_cnt_t := before_cnt_t - 1;
                if (before_cnt_t = 0 ∧ t ∈ wait_trset)
                then unblock t;
                endif
            endif
        enddo
   _3>
  ).
```

After a transaction commits, all the transactions in its *before_trset* need to be aborted because they must commit, if they can, before this transaction. The critical section of class 1 in the procedure guarantees that transactions cannot switch into the write phase concurrently, and once a transaction is committed and assigned a final-timestamp, no transaction in its *before_trset* can commit. Note that LP1 is also in the critical section of the same class. This achieves mutual exclusion on *before_cnt* and *write_trset*. The critical section of class 3 in the procedure has the same effect as that of the critical section in the procedure *tabort*.

## 2.3. Write Phase

Once a transaction is in the write phase, it is considered to be committed. All committed transactions can be serialized by the final-timestamp order. In the write phase, the only work of a transaction is making all its updates permanent in the database. Data items are copied from the local workspace into the database. After each write operation, the corresponding write lock is released. The Thomas' Write Rule (TWR) is applied here. The write requests of each transaction are sent to the data manager, which carries out the write operations in the database. Transactions submit write requests along with their final-timestamps. The write procedure is as follows:

```
twrite = (
  <_4 for x ∈ { x_i | wlock (id,x_i) } do
          for T ∈ write_trset do
            if ( wlock(T,x) ∧ ts(T) < ts(id) )
            then release T's write lock on x;
            endif
          enddo
         send write request on x and wait for
         acknowledgement;
  _4>
         if (acknowledgement is ok)
         then release the write lock on x;
         else  abort;
         endif
     enddo
     R_Q := R_Q - {id};
  ).
```

The purpose of the critical section is to achieve mutual exclusion on write locks. For each data object, write requests are sent to the data manager only in ascending timestamp order. After a write request on data object $x$ with timestamp $n$ is issued to the data manager, no other write request on $x$ with a timestamp smaller that $n$ will be sent. The write requests are buffered by the data manager. The data manager can work with the first-come-first-serve policy or always select the write request with the highest priority to process. When a new request arrives, if there is another buffered write request on the same data object, the request with the smaller timestamp is discarded. Therefore for each data object there is at most one write request in the buffer. This, in conjunction with the procedure *twrite*, guarantees TWR.

## 3. CPU Scheduling

Although the focal point of this paper is on concurrency control, i.e. operation level scheduling, we still need to discuss a little about the transaction scheduling, or CPU scheduling, aspect of our algorithm. In non-real-time database systems, CPU scheduling is usually done by the underlying operating systems, because there are no timing constraints. Data consistency is the only concern. In real-time database systems, however, CPU scheduling should take into account the timeliness of transactions.

In our protocol, R_Q contains all transactions that can be scheduled. These transactions can be in any phase. We need a policy to determine the CPU scheduling priority for transactions in different phases. Transactions in their wait phase are those that have finished their work and are waiting for their chances to commit. We would like to avoid aborting such transactions as much as possible. Therefore transactions in this phase are given higher CPU scheduling priority than those in the read phase so that they can commit as soon as they get the chance. Transactions in the read phase are scheduled according to their assigned priority. If there are several read phase transactions in the $R\_Q$, the one with the highest priority is always selected to execute.

For transactions in the wait phase, the lower the priority is, the higher the CPU scheduling priority is. Since low priority transactions are more vulnerable to conflicts, if there is a chance, they should be committed as soon as possible to avoid being aborted later. Moreover, when a high priority transaction $T_H$ is committed, it may have to abort a low priority transaction $T_L$ if $T_L$ is in $T_H$'s *before_trset*. If $T_L$ is also ready to commit and we allow it to commit before $T_H$, both $T_L$ and $T_H$ can be committed.

## 4. Correctness of the Algorithm

In this section, we give an informal argument on the correctness of the algorithm. We will also show that the algorithm is free from deadlocks. First, we give the simple definitions of *history* and *serialization graph* (SG). For the formal definitions, see [Bern87]. A history is a partial order of operations that represents the execution of a set of transactions. Any two conflicting operations must be comparable. Let $H$ be a history. The *serialization graph* for $H$, denoted by SG($H$), is a directed graph whose nodes are committed transactions in $H$ and whose edges are all $T_i \rightarrow T_j$ $(i \neq j)$ such that one of $T_i$'s operations precedes and conflicts with one of $T_j$'s operations in $H$. To prove a history $H$ serializable, we only have to prove that SG($H$) is acyclic [Bern87].

Let $T_1$ and $T_2$ be two committed transactions in a history $H$ produced by the algorithm. We argue that if there is an edge $T_1 \rightarrow T_2$ in SG($H$), then $ts(T_1) < ts(T_2)$. Since $T_1 \rightarrow T_2$, The two must have conflicting operations. There are three cases.

Case 1: $w_1[x] \rightarrow w_2[x]$

Suppose $ts(T_2) < ts(T_1)$. Therefore $T_2$ enters into the write phase before $T_1$. If $w_1[x]$ is sent to the data manager first, $T_2$'s write lock on $x$ must be released before $w_1[x]$ is sent to the data manager in *twrite()*. If $w_2[x]$ is sent to the data manager first, it will either be processed before $w_1[x]$ is sent to the data manager, or be discarded when the data manager receives $w_1[x]$, because $w_2[x]$ has a smaller timestamp. Therefore $w_1[x]$ is never processed before $w_2[x]$. Such conflict is impossible. A contradiction.

Case 2: $r_1[x] \rightarrow w_2[x]$

If $T_2$ holds write lock on $x$ when $T_1$ requests the read lock, we must have $priority(T_1) > priority(T_2)$ and $T_2$ is not in the write phase, because otherwise $T_1$ would have been blocked by LP1. By LP1, $T_2 \in after\_trset(T_1)$. $T_2$ will not switch into the write phase before $T_1$ does, because $before\_cnt_{T_2}$ cannot be zero with $T_1$ still in the read or wait phase. Therefore $ts(T_1) < ts(T_2)$. If $T_1$ holds read lock on $x$ when $T_2$ requests the write lock, by LP2, we have either $T_2 \in after\_trset_{T_1}$ or $T_1 \in before\_trset_{T_2}$, depending on the priorities of the two transactions. In either case, $T_1$ must commit before $T_2$. Hence we also have $ts(T_1) < ts(T_2)$.

Case 3: $w_1[x] \rightarrow r_2[x]$

Since $T_1$ is already in the write phase before $T_2$ reads $x$, we must have $(T_1) < ts(T_2)$.

Suppose there is a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$ in SG($H$). By the above argument, we have $ts(T_1) < ts(T_2) < \cdots < ts(T_n) < ts(T_1)$. This is impossible. Therefore no cycle can exist in SG($H$) and thus the algorithm only produces serializable histories.

In the algorithm, a high priority transaction can be blocked by a low priority transaction only if the low priority transaction is in the write phase. Suppose there is a cycle in the wait-for graph (WFG), $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$. For any edge $T_i \rightarrow T_j$ in the cycle, if $priority(T_i) > priority(T_j)$, $T_j$ must be in the write phase, thus it cannot be blocked by any other transactions and cannot appear in the cycle. Therefore we must have $priority(T_i) < priority(T_j)$ and thus $priority(T_1) < priority(T_2) < \cdots < priority(T_n) < priority(T_1)$. This is impossible. Hence a deadlock cannot exist.

The strictness of the histories produced by the algorithm follows obviously from the fact that a transaction applies the results of its write operations from its local workspace into the database only after it commits. This property makes transaction recovery procedure simpler than non-strict concurrency control algorithms.

## 5. An Example

In this section, we give a simple example to show how the algorithm works. The example is depicted in Fig. 3. A solid line at a low level indicates that the corresponding transaction is doing I/O operation due to a page fault or in the write phase. A dotted line at a low level indicates that the corresponding transaction is either suspended or blocked, and not doing any I/O operation either. A line raised to a higher level indicates that the transaction is executing. The absence of a line indicates that the transaction has not yet arrived or has already completed.

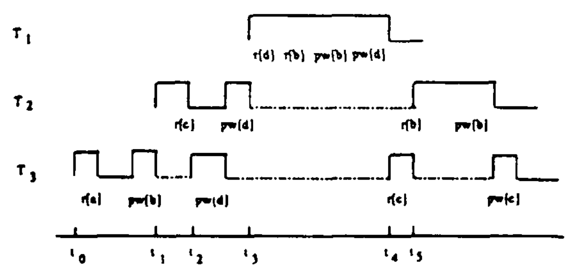


Fig. 3 An Example

There are three transactions in the example. $T_1$ has the highest priority and $T_3$ has the lowest. $T_3$ arrives at time $t_0$ and reads data object $a$. This causes a page fault. After the I/O operation, it pre-writes $b$. Then $T_2$ comes in at time $t_1$ and preempts $T_3$. At time $t_2$ it reads $c$ and causes another page fault. So it is blocked for the I/O operation and $T_3$ executes. After $T_3$ pre-writes $d$, $T_2$ finishes I/O and preempts $T_3$ again. It pre-writes $d$ which is only write locked by $T_3$. At time $t_3$, $T_1$ arrives and preempts $T_2$. $T_1$ first reads $d$, which is write locked by both $T_2$ and $T_3$. Therefore, $before\_trset_{T_1}$ becomes $\{T_2, T_3\}$ and both $before\_cnt_{T_2}$ and $before\_cnt_{T_3}$ become 1. Then $T_1$ reads $b$, which is write locked by $T_3$. Since $T_3$ is already in $before\_trset_{T_1}$, nothing is changed. Then $T_1$ pre-writes $b$ and pre-writes $d$. Since these two data objects are not read locked by any other transactions, the write locks are granted to $T_1$ directly. At time $t_4$, $T_1$ switches into the write phase. Both $before\_cnt_{T_2}$ and $before\_cnt_{T_3}$ go back to 0. Now $T_2$ should be executed, but it needs to read $b$, which is being write locked by $T_1$; hence $T_3$ is executed instead. It reads $c$, which is read locked by $T_2$. At time $t_5$, $T_1$ finishes writing $b$ and releases the write lock so that $T_2$ can preempt $T_3$ to continue its work. It reads $b$, which is write locked by $T_3$. Now $before\_trset_{T_2}$ becomes $\{T_3\}$ and $before\_cnt_{T_3}$ becomes 1. After $T_2$ pre-writes $b$, it switches into the write phase and $before\_cnt_{T_3}$ becomes 0 again. Then $T_3$ executes and also switches into write phase after pre-writing $c$.

In the above example, $T_1$, which is supposed to be the most urgent transaction, finishes first although it is the last to arrive. $T_3$, which is supposed to be the least urgent one, is the last one to commit. None of the three transactions need to be aborted. Assume we use 2PL in the above example. When a high priority transaction requests a lock which is held by a low priority transaction, we either let the high priority transaction to wait or abort the low priority transaction. Suppose we choose the first alternative, both $T_1$ and $T_2$ would be blocked by $T_3$ because $T_3$ holds a write lock on $d$. If we choose the second alternative, $T_3$ will be aborted by $T_2$ when $T_2$ pre-writes $d$ and then $T_2$ will be aborted by $T_1$ when $T_1$ reads $d$. This example illustrates the advantage of the proposed method over 2PL.

## 6. Conclusions

Time-critical scheduling in real-time database systems consists of two scheduling mechanisms: transaction scheduling and operation scheduling. To find new concurrency control methods in which timing constraints of transactions are taken into account, we have investigated solutions to the operation scheduling aspect of time-critical scheduling.

In this paper, a priority-based concurrency control method for real-time database systems is presented which employs a priority-dependent locking mechanism. It works under the condition that no information about data requirements or execution time of each transaction is available. By delaying the write operations of transactions, the restraint of past transaction execution on the serialization order is relaxed, allowing the serialization order among transactions to be adjusted dynamically in compliance with transaction timeliness and criticality. The new algorithm features the ability that allows transactions to meet the timing constraints as much as possible without reducing the concurrency level of the system or increasing the restart rate significantly. In the algorithm, high priority transactions are never blocked by an uncommitted lower priority transaction, while low priority transactions may not have to be aborted even in face of conflict with high priroity transactions. In conjunction with a time-critical transaction scheduling policy (CPU scheduling policy) discussed in Section 3, the proposed algorithm is expected to improve the system performance significantly.

## References

[Abb88] R. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: a performance evaluation," in *Proc. 14th VLDB Conf.*, Los Angeles, Aug. 1988.

[Bern87] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1987.

[Bok87] C. Boksenbaum et al., "Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems," *IEEE Trans. on Software Eng.*, vol. SE-13, no. 4, April 1987, pp 409-419.

[Buch89] A. Buchmann et al., "Time-critical database scheduling: a framework for integrating real-time scheduling and concurrency control," in *Proc. Data Engineering Conf.*, Los Angeles, Feb. 1989.

[Gar83] H. Garcia-Molina, "Using semantic knowledge for transaction processing in a distributed database," *ACM Trans. Database Syst.*, vol. 8, no. 2, pp. 186-213, June 1983.

[Hua89] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham, "Experimental evaluation of real-time transaction processing," in *Proc. Real-time Systems Symp.*, Dec. 1989.

[Kor90] H. Korth, "Triggered Real-Time Databases with Consistency Constraints," *16th VLDB Conference*, Brisbane, Australia, Aug. 1990.

[Kung81] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213-226, June 1981.

[Lin89] K. J. Lin, "Consistency issues in real-time database systems," in *Proc. 22nd Hawaii Intl. Conf. System Sciences*, Hawaii, Jan. 1989.

[Sha88] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Concurrency control for distributed real-time databases," *ACM SIGMOD Record*, vol. 17, no. 1, Mar. 1988.

[Sha90] L. Sha, R. Rajkumar, S. H. Son, and C. Chang, "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, to appear.

[Son88] S. H. Son, editor, *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988.

[Son89] S. H. Son and R. P. Cook, "Scheduling and consistency in real-time database systems," in *Proc. 6th IEEE Workshop Real-Time Operating Systems and Software*, Pittsburgh, Pennsylvania, May 1989, pp 42-45.

[Son89b] S. H. Son, "On Priority-Based Synchronization Protocols for Distributed Real-Time Database Systems," *IFAC/IFIP Workshop on Distributed Databases in Real-Time Control*, Budapest, Hungary, Oct. 1989, pp 67-72.

[Son90] S. H. Son and C. Chang, "Performance evaluation of real-time locking protocols using a distributed software prototyping environment," to appear in *Proc. 10th Intl. Conf. Distributed Computing Syst.*, Paris, France, June 1990, pp 124-131.

[Son90b] S. H. Son and J. Lee, "Scheduling Real-Time Transactions in Distributed Database Systems," *7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, Virginia, May 1990, pp 39-43.

[Son90c] S. H. Son, "Real-Time Database Systems: A New Challenge," *Data Engineering, vol. 13, no. 4*, Special Issue on Future Directions on Database Research, Dec. 1990.

[Stan88] J. A. Stankovic and W. Zhao, "On real-time transactions," *ACM SIGMOD Record*, vol. 17, no. 1, Mar. 1988.

[Vrb88] S. V. Vrbsky and K. J. Lin, "Recovering imprecise transactions with real-time constraints," in *Proc. Symp. Reliable Distributed Syst.*, Oct. 1988, pp. 185-193.

# New Paradigms for Real-Time Database Systems

Robert P. Cook, Sang H. Son, Henry Y. Oh, Juhnyoung Lee

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

## 1. Introduction

Real-time database systems (RTDBS) are database systems where transactions have timing constraints such as *deadlines*. The correctness of the system depends not only on the logical results but also on the time within which the results are produced. In RTDBS, transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query in the tracking data for a mission must be processed within given deadlines.

Conventional database systems are typically not used in real-time applications due to poor performance and lack of predictability. In other words, paradigms used in conventional database systems are not suitable in real-time database systems [Son90]. To address this problem, we have been investigating new database technology and paradigms for real-time systems using both theoretical as well as experimental approaches. They can be grouped into the following research tasks: (1) investigating new protocols for transaction scheduling, concurrency control, and checkpointing, and (2) developing experimental database systems that can provide real-time features over conventional relational databases. New scheduling and concurrency control protocols developed in the first task are being implemented in the experimental database systems and the prototyping environment for performance evaluation.

Our research effort in the area of real-time transaction scheduling has resulted in two new protocols: one based on locking [Lin90] and the other on timestamp ordering. In the area of experimental database systems, we have been developing a suite of database systems on several platforms. Currently, our research utilizes the UNIX, StarLite [Cook90], and ARTS operating systems [Tok89]. Experimental database systems we have developed on these platforms are the Multi-user Real-time Database (MRDB), Parallel Real-time Database (PRDB), and Real Time Database (RTDB), respectively [Son91]. All three systems are based on the relational paradigm. Much of our development consists of implementing new functionality on the most appropriate platform, and where applicable, porting the result to one of the others. In this paper, we outline the scheduling protocol based on timestamp ordering and our experience with PRDB development.

## 2. An Optimistic Concurrency Control for Real-Time Transaction Scheduling

In real-time transaction scheduling, the actual execution order of operations is determined by two factors: priority order and serialization order among transactions in system. The difficulties in real-time transaction scheduling arise from the fact that these two factors have different natures and are constructed in different ways. While serializable execution order is strictly bound to the past execution history, the priority order does not reflect the past execution history and may dynamically destroy the order set up in the past execution, hence serializability. By identifying the effects of the interactions between serialization order and priority order in scheduling real-time transactions, we can build more intelligent conflict resolution schedulers.

One approach to real-time transaction scheduling is to make the priority order and serialization order compatible as much as possible in order to increase the probability of satisfying both timing and

This work was supported in part by ONR contract # N00014-88-K-0245, by NOSC, and by IBM FSD.

consistency constraints. One way to make the two orders compatible is to adjust serialization order dynamically to priority order. This approach can be justified because serialization order is not subject to timing constraints as long as it enforces serializability, while we assume that the priority order of a transaction is statically determined when it arrives in the system.

Integrating a concurrency control protocol with priority-based scheduling methods has the inherent disadvantage of being limited by the concurrency control protocol on which it depends. Two-phase locking and timestamp ordering depend on the immediate validation of operations, and do not provide a facility to adjust serialization order dynamically to priority order. To adjust the serialization order, we need to delay determining the serialization order of conflicting operations, because once the serialization order is determined, the orders of operations from transactions cannot be adjusted dynamically.

In optimistic concurrency control in which the serializability test (called the *validation test*) is made only at the end of a transaction, the serialization order can be constructed dynamically in compliance with transaction timeliness and criticality. Furthermore, owing to its potential for a high degree of parallelism, optimistic concurrency control is expected to perform better than two-phase locking or timestamp ordering in real-time transaction scheduling.

We have developed an optimistic concurrency control protocol based on the notion of dynamic timestamp allocation [Bok87]. In this protocol, the serialization order is dynamically constructed by using intervals of timestamps. The protocol uses a *backward validation* scheme, in which validating a transaction is performed against committed transactions. It also updates the timestamp intervals of active transactions to adjust their serialization order. As in other optimistic protocols, the execution of a transaction in our protocol is divided into three phases: read, validation, and write. However, unlike other optimistic protocols, conflicts and nonserializable executions are detected during the read phase of transaction execution, minimizing wasted work due to later restarts of transactions.

The goal of this protocol is to enforce serializability by satisfying the following two conditions (C1) and (C2) through every read, prewrite, and validation. As long as (C1) and (C2) are satisfied, serialization order can be adjusted in favor of priority order without violating data consistency.

(C1) Each timestamp interval constructed when a transaction accesses a data object should preserve the order induced by the timestamps of all committed transactions which have accessed that data object.

(C1) The order induced by timestamp values of a validating transaction should not destroy the serialization order constructed by the past execution, i.e., by committed transactions.

Before describing the algorithms for the read and validation phases, we summarize the information used to keep track of the dependencies among transactions:

- for each active transaction $T$, its readset, $RS(T)$, and writeset, $WS(T)$;
- for each committed transaction $T$, a timestamp $ts(T)$ assigned in its validation phase;
- for each active transaction $T$ and for each data object $x$ it has read or written in its read phase, an interval of timestamps $I(T,x)$; and
- for each data object $x$, $RTS(x)$ and $WTS(x)$, which denote the largest timestamps of the committed transactions having read or written $x$, respectively.

In order to decide whether a transaction $T$ is involved in a nonserializable execution, all the timestamp intervals of $T$ are grouped as $I(T) = \bigcap_{x \in X} I(T,x)$ for X being the set of data objects accessed by $T$. $I(T)$ preserves the order between $T$ and committed transactions. Any operation of an active transaction $T$ which introduces a nonserializable execution can be detected by checking whether the execution of the operation results in $I(T) = \varnothing$.

In the implementation, with each transaction $T$ is associated its *current interval* $I_c(T)$ instead of $I(T,x)$'s and $I(T)$. At the start of $T$, $I_c(T)$ is initialized as $[0, \infty)$ (the whole set of allowable timestamps). For each read or prewrite made by $T$, $I_c(T)$ is adjusted according to dependencies induced by the operation to satisfy (C1). A transaction $T$ must be restarted when $I_c(T) = \emptyset$. The gradual construction of a serialization order by using $I_c(T)$ makes it possible to detect nonserializable executions even before the transaction reaches its validation phase. Furthermore, every transaction that reaches its validation phase is guaranteed to commit in this protocol.

We present the protocol via the following pseudo code. We bracket a critical section by "<" and ">", and assume that timestamp intervals contain only integers.

```
Read phase

< for every data object $x$ in $RS(T_i)$ do
    $I_c(T_i) := I_c(T_i) \cap [WTS(x)+1, \infty)$ >
    if $I_c(T_i) = \emptyset$ then restart($T_i$)

< for every data object $x$ in $WS(T_i)$ do
    $I_c(T_i) := I_c(T_i) \cap [WTS(x)+1, \infty) \cap [RTS(x)+1, \infty)$ >
    if $I_c(T_i) = \emptyset$ then restart($T_i$)

Validation and Write phase

< choose $ts(T_i)$ in $I_c(T_i)$
  update $RTS(x)$ and $WTS(x)$ for every $x$ in $RS(T_i)$ and $WS(T_i)$
  adjust $I_c(T_j)$ >
  make its updates permanent in the database
```

The validation of a transaction means that the execution of the operations from the transaction is serializable, and the execution should be reflected in the serialization order of committed transactions. Thus we should choose a timestamp for the transaction to satisfy (C2), update $RTS$ and $WTS$ for data objects it accessed, if necessary, and adjust the timestamp intervals of all active transactions which conflict with it to satisfy (C1). Any timestamp $ts \in I_c(T_i)$ satisfies the condition (C2). The adjustment procedure is as the following:

```
Interval Adjustment Operation

< for every data object $x$ in $RS(T_i)$ do
    for every transaction $T_j$ which has written $x$ do
        $I_c(T_j) := I_c(T_j) \cap [ts(T_i)+1, \infty)$ >
        if $I_c(T_j) = \emptyset$ then restart($T_j$)

< for every data object $x$ in $WS(T_i)$ do
    for every transaction $T_j$ which has read $x$ do
        $I_c(T_j) := I_c(T_j) \cap [0, ts(T_i)-1]$
    for every transaction $T_j$ which has written $x$ do
        $I_c(T_j) := I_c(T_j) \cap [ts(T_i)+1, \infty)$ >
        if $I_c(T_j) = \emptyset$ then restart($T_j$)
```

The Adjust procedure given above can be modified in several ways to integrate priority scheduling with this protocol. As a simple approach, we can adjust the size of $I_c(T_j)$ of an active transaction $T_j$. Because the size is correlated with the probability of restarting of the transaction, for priority scheduling, a transaction with higher priority needs to have a larger timestamp interval than a transaction with lower priority. When adjusting the timestamp intervals of active transactions, if we give larger timestamp intervals to transactions with higher priority over transactions with lower priority, then we can decrease the risk of restarting higher priority transactions. The choice of a timestamp of the validating transaction also has a definite effect on the active transactions which conflict with it, because the timestamp intervals of those transactions are adjusted according to the timestamp chosen.

As another approach, the *priority wait* strategy [Har90] in which the validating transaction waits for the conflicting transactions with higher priority to complete, can also be used in this protocol. The advantage of this strategy is that a higher priority transaction is not restarted due to the validation of a lower priority transaction. While a lower priority transaction is waiting, it is possible that it will be restarted due to the validation of one of the conflicting higher priority transactions.

## 3. A New Parallel Paradigm for Real-Time Database System

One important advance in computing technology is the emergence of parallel computers. In a database system, there are at least two levels in which parallelism can be exploited. The first level contains the basic database operations. The basic idea behind these algorithms is to partition a single database operation into multiple sub-operations, perform those sub-operations simultaneously and then combine the separate results into one. For example, the join operation can be performed in parallel by dividing one of the two relations into several blocks and joining each block with the other relation simultaneously. As a large amount of data are usually involved in each database operation, it is essential from a performance standpoint that accessing the data should be done efficiently. New techniques to organize indices and to structure data files are needed.

The second level is the query processing level in which different queries can be executed simultaneously if they do not conflict. For example, two CREATE operations can be executed in parallel on different processors or the interpretation of two expressions can be done simultaneously. Here we are only concerned with parallelism at the second level.

PRDB is an experimental real-time database system that runs on an emulated tightly-coupled, shared-memory multiprocessor system in the StarLite software development environment, running on UNIX under SunView/X Windows. The overall design goal of PRDB is to provide a general paradigm for exploring parallelism and implementing different real-time scheduling policies in database systems. The paradigm has evolved from the WorkCrew model [Rob89]. The major advantage of the WorkCrew paradigm is its efficient mechanisms to control and manage parallelism by creating the minimum number of processes in the system and the employment of a lazy evaluation technique for posted work. The synchronization of concurrent tasks and the overhead of task decomposition are minimized.

In the WorkCrew paradigm, tasks are assigned to a finite set of workers. A task may consist of several subtasks. If some of the subtasks can be executed in parallel, they are put into a "request_help" queue of the worker. Any idle worker can take over the subtasks and execute them. The WorkCrew paradigm has two advantages. First, much of the work associated with task division can be deferred until a new worker actually undertakes the subtask, and avoided altogether if the original worker ends up executing the subtask serially. Second, the number of active workers in the system is always equal to the number of processors.

However, the WorkCrew paradigm has two limitations that prevent it from becoming a general framework for parallel computing. The first limitation is that there is no general mechanism to retrieve results. In the WorkCrew model, the results of operations are reflected in the preallocated space. If operations produce some new results apart from the results stored in preallocated space, which is usually the case for most of the applications, there is no way to retrieve those results. The second limitation is that there is no way to specify different operations to be performed on data, i.e., the procedure to manipulate a set of data cannot be explicitly passed to each worker so that the worker can perform different operations. Further, the WorkCrew model does not address the real-time requirements of the application.

In our paradigm, the first limitation is addressed by providing a result queue for the crew. The second limitation is dealt with by passing the handler for operations as a parameter to each worker. These improvements require the extension of the concept of work. The concept of work in the WorkCrew paradigm is a passive entity and consists only of the data items to be manipulated. In the PRDB paradigm, the concept of work is still a passive entity, however, the contents of work not only consist of data items to be manipulated, but also the operation to be performed on the data items and the timing-constraint information for the work to be performed.

The real-time transaction scheduler and the CPU schedulers (called *dispatchers)* are separated. The real-time transaction scheduler is implemented by the crew, while the dispatcher is implemented within each worker. The real-time transaction scheduler schedules tasks according to its own policies and puts them onto two work queues residing on the crew. One of these two queues is for hard deadline tasks and the other is for soft deadline tasks. Since each worker has also its own "request_help" queue, the search path of work to do by an idle worker begins with the hard-deadline queue of the crew, then the "request_help" queues of the workers, and finally the soft deadline queue of the crew. If the deadline has passed, the workers immediately write the result into the result queue indicating the missing of a deadline. Otherwise, the work is performed and results are returned through the result queue. In the case where a worker has to synchronize with other workers in performing a task, the worker blocks and a new worker is created to help the other workers' work. Thus, the number of the active workers is always equal to that of the processors in the system, if the work load is high.

The data structures of a unit of work and a unit of result are as follows:

```
WORK = RECORD
            critical : CARDINAL; (* hard vs soft deadline *)
            deadline : Time; (* the deadline is checked before executing the operation *)
            operation : PROCEDURE; (* specifying the operation *)
            paramAddr : ADDRESS; (* pointer to the work to be done *)
            size : CARDINAL; (* the size of the work data structure *)
END;
RESULT = RECORD
            missDeadline : BOOLEAN; (* missed deadline? *)
            finishTime : Time; (* the finished time of a unit of work *)
            resultAddr : ADDRESS; (* pointer to the result data structure *)
            size : CARDINAL; (* the size of the result data structure *)
END;
```

The major functions provided by the paradigm are starting a crew of workers, destroying a crew of workers, modifying the number of workers in a crew, assigning work to a crew, requesting help by a worker, testing whether the requested work has been done by other workers, and waiting for some work to be finished.

Each basic database operation is written by using the functions provided above if some part of the basic database operation can be done in parallel. Initial results have indicated the soundness of the paradigm for parallel real-time database computing. More thorough experiments are being carried out. We believe that this new paradigm will scale well to large number of processors in the system and will be efficient in scheduling real-time transactions.

The data given below are the relative speedups of PRDB over the RDB system. The workload for the experiments is the same for the uniprocessor which runs the RDB system and the multiprocessor system which runs PRDB. The first experiment (Test1) consists of 26 "Create" operations and 22 "Insert" operations. Each "Insert" operation inserts 15 tuples in a different relation with three attributes each. Other experiments (Tests 2 and 3) consist of the same operations as Test1, however, each "Insert" operation in Test2 inserts 25 tuples, while each "Insert" operation in Test3 consists of 50 Tuples. The results show that PRDB favors coarse-grained parallelism in the computation.

| Speedup of PRDB over RDB | | | | | | | |
|---|---|---|---|---|---|---|---|
| Number of processors | | 1 | 2 | 3 | 4 | 5 | 6 |
| Test1 | Time Units | 4613 | 3704 | 3074 | 2593 | 2515 | 2447 |
| | Speedup | | 1.24 | 1.50 | 1.77 | 1.83 | 1.88 |
| Test2 | Time Units | 9046 | 5761 | 4170 | 3471 | 3120 | 2904 |
| | Speedup | | 1.57 | 2.16 | 2.60 | 2.89 | 3.11 |
| Test3 | Time Units | 26195 | 14276 | 9878 | 7813 | 6752 | 5841 |
| | Speedup | | 1.83 | 2.65 | 3.35 | 3.87 | 4.48 |

## 4. Concluding Remarks

A real-time database manager is one of the critical components of a real-time system. To satisfy timing requirement, transactions must be scheduled considering not only the consistency constraints but also their timing constraints. In addition, the system should support a predictable behavior such that the possibility of missing deadlines of critical tasks could be informed ahead of time, before their deadlines expire. In this paper, we have presented new paradigms that exploit the ideas of dynamic adjustment of serialization order and parallel computing. We are currently working on the performance evaluation of new paradigms using the prototyping environment as well as experimental database systems.

## REFERENCES

[Bok87] C. Boksenbaum, M. Cart, J. Ferrie, and J. Pons, "Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 4, April 1987.

[Cook90] R. Cook, and Y. Oh, "The StarLite Project," *The 3rd Sym. on Frontiers of Massively Parallel Computation*, Univ. of Maryland,College Park, Oct.1990.

[Har90] J.R. Haritsa, M.J. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[Lin90] Y. Lin and S. H. Son, "Concurrency Control in Real-Time Database Systems by Dynamic Adjustment of Serialization Order," *IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[Rob89] E. S. Roberts, and M. T. Vandevoorde, "WorkCrews: An Abstraction for Controlling Parallelism," *DEC SRC Technical Report*, April 1989.

[Son90] S. H. Son, "Real-Time Database Systems: A New Challenge," *Data Engineering*, vol. 13, no. 4, Special Issue on Directions for Future Database Research and Development, December 1990.

[Son91] S. H. Son, M. Poris, and C. Iannacone, "Implementing a Distributed Real-Time Database Manager," *The Second International Symposium on Database Systems for Advanced Applications (DASFAA '91)*, Tokyo, Japan, April 1991.

[Tok89] H. Tokuda and C. Mercer, ARTS: A Distributed Real-Time Kernel, ACM Operating Systems Review, 23 (3), July 1989.

# RTDB: A Real-Time Database Manager for Time-Critical Applications

Sang H. Son, Carmen C. Iannacone, and Marc S. Poris

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
USA

## Abstract

Compared with traditional databases, database systems for time-critical applications have the distinct feature that they must satisfy timing constraints associated with transactions. Transactions in real-time database systems should be scheduled considering both data consistency and timing constraints. Since a database system must operate in the context of available operating system services, an environment for database systems development must provide facilities to support operating system functions and integrate them with database systems for experimentation. We chose the ARTS real-time operating system kernel. In this paper, we present our experience in integrating a relational database manager with a real-time operating system kernel, and our attempts at providing flexible control for concurrent transaction management. Current research issues involving the development of a programming interface and our efforts in using these techniques in implementing a specific experimental application are also discussed.

## 1. Introduction

Real-time computing is an open research area [Stan88]. The growing importance of real-time computing in a large number of applications, such as aerospace and defense systems, industrial automation and robotics, and nuclear power plants, has resulted in an increased research effort in this area. Recent workshops have pointed to the need for basic research in database systems that satisfy timing constraints in collecting, updating, and retrieving shared data, since traditional data models and databases are not adequate for real-time systems [IEEE90, ONR90]. Very few conventional database systems allow users to specify timing constraints or ensure that the system meets those set by the user. Interest in this new application domain is also growing in the database community. Recently, a number of research results have appeared in the literature [Abb89, Buc89, Kor90, Lin90, Sha88, Sha91, Son88, Son89, Son90].

Time is the key factor to be considered in real-time database systems, and the correctness of the system depends not only on the logical results but also on the time within which the results are produced. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on the tracking data for a missile must be processed within given deadlines, satisfying not only database consistency constraints but also timing constraints.

Conventional database systems are typically not used in real-time applications due to the inadequacies of poor performance and lack of predictability. Current database systems do not schedule their transactions to meet response requirements and they commonly lock data tables to assure only the consistency of the database. Locks and time-driven scheduling are basically incompatible, resulting in response requirement failures when low priority transactions block higher priority transactions. New techniques are required to manage the consistency of real-time databases, and they should be compatible with time-driven scheduling and meet both the required system response predictability and temporal consistency.

To address the inadequacies of current database systems, the transaction scheduler needs to be able to take advantage of the semantic and timing information associated with data objects and transactions. A model of real-time transactions needs to be developed which characterizes distinctive features of real-time databases that can contribute to the improved responsiveness of the system. The semantic information of the transactions investigated in the modeling study can be used to develop efficient transaction schedulers [Son90b,

This work was supported in part by ONR under contract N00014 88-K-0245, by the Naval Ocean Systems Center, and by IBM Federal Systems Division.

Son91].

A database system must operate in the context of available operating system services, because correct functioning and timing behavior of database control algorithms depend on the services of the underlying operating system. As pointed out by Stonebraker, operating system services in many systems are not appropriate for support of database functions [Ston81]. In many areas, such as buffer management, recovery, and consistency control, operating system facilities have to be duplicated by database systems because they are too slow or inappropriate. An environment for database systems development must, therefore, provide facilities to support operating system functions and integrate them with database systems for experimentation.

The ARTS real-time operating system kernel, under development at Carnegie-Mellon University, attempts to provide a "predictable, analyzable, and reliable distributed real-time computing environment" which is an excellent foundation for a real-time database system [Tok89]. The ARTS system, which provides support for programs written in C and C++, implements different prioritized and non-prioritized scheduling algorithms and prioritized message passing as well as supporting lightweight tasks. All of these features are important when considering a real-time database.

Our research effort resulted in a new relational database manager for distributed real-time systems. We have used the relational database technology since it provides the most flexible means of accessing distributed data. In this paper, we present our experience in integrating a relational database manager with a real-time operating system kernel, and our attempts at providing flexible control for concurrent transaction management using a technique called *workload mediation*. Current research issues involving the development of a programming interface, associated issues of client/server object development which can be simplified through the use of templates, and our efforts in using these techniques in implementing a specific experimental application are also discussed.

## 2. The ARTS Real-Time OS Kernel

Research in the area of distributed, real-time operating systems indicates that most are designed for a specific need, and as such are difficult to build, maintain, and modify; in addition, they do not afford the capability of predicting runtime behavior during application design. In fact, few non-real-time operating systems provide a functionally complete set of general purpose, real-time task and time management functions, despite the fact that the user community is expressing the desire for increasingly complex applications of this type. Since the success of applications in real-time computing is primarily contingent on a system's temporal functionality, what is needed is an environment in which the system engineer can analyze and predict, during the design stage, whether the given real-time tasks having various types of system and task interactions (i.e. memory allocation/deallocation, message communications, I/O interactions, etc.) can meet their timing requirements.

In an attempt to provide such functionality, ARTS provides the process and data encapsulation that other distributed, object-oriented operating systems do, while at the same time including elements of temporal significance to the services it provides. This integration of data, thread and concurrency control greatly facilitates real-time schedulability analysis. The ARTS can support both hard and soft real-time tasks as well as periodic and sporadic ones [Tok89].

To support time-critical operations, the ARTS programming language interface allows designers to specify timing requirements and the chosen communication structure so that they are visible at both the language and system level; this allows the system-wide ARTS environment to make scheduling decisions based on both temporal constraints and priority. The Integrated Time-Driven Scheduler (ITDS) model of the ARTS is more effective than the common priority-based preemptive scheduling of many real-time systems. Such simple schedulers become confused during heavy system loads when they cannot decide which tasks are important and should be completed and which tasks should be aborted, causing unpredictability in the applications. The ITDS model however, employs a time-varying "value function" which specifies both a task's time criticality and semantic importance simultaneously. A hard real-time task can be characterized by a step function where the discontinuity occurs at the deadline, while soft real-time tasks are described by continuous (linear or non-linear) decreasing function after its critical time. In addition, ARTS' designers have separated the policy and mechanism layers, so that users can implement new scheduling policies with a minimum of effort, even dynamically changing the policy during runtime.

The issue of priority inversion is crucial to providing semantically correct system behavior in addition to addressing temporal concerns. Priority inversion occurs when a high priority activity waits for a lower priority activity to complete. Resource sharing and communication among the executing tasks can lead to priority inversion if the operating system does not manage the available resource set properly. Significant research in the construction of ARTS was done to avoid priority inversion among concurrently executing tasks; in the processor scheduling domain, low priority servers which provide service to clients of all priorities are susceptible to inversion. For example, when a low priority request is being serviced, a high priority task requests the same service; since the server's computation is non-preemptable, the high priority request waits. Any task of higher priority than the server may preempt the server itself, however, so if a medium priority task arrives it preempts the server indefinitely, causing the high priority job to be lost in the shuffle. The ARTS employs a priority inheritance mechanism to propagate information about a single computation which crosses task boundaries. That is, if a server task accepts the request of a client, the server inherits the priority of the client. Furthermore, the server should also inherit the priority of the highest priority task waiting for the service.

The notion of time encapsulation cannot be divorced from the basic structure of ARTS, in which every computational entity is represented as an object, called an *artobject*. An artobject is defined as either a passive or an active object. In a passive object, there is no explicit declaration of a thread which accepts incoming invocation requests while an active object contains one or more threads defined by the user. In an active object, its designer is responsible for providing concurrency control among coexecuting operations. When a new instance of an active object is created, its root thread will be created and run immediately. A thread can create threads within its object.

The ARTS kernel supports the notion of real-time objects and real-time threads. A real-time object is defined with a "time fence," a timer associated with the thread which ensures that the remaining slack time is larger than the worst case execution time for the operation. A real-time thread can have a value function and timing constraints related to its execution period, worst case execution time, phase, and delay value. When an operation with a time fence is invoked, the operation will be executed (or accepted) if there is enough remaining computation time against the specified worst case execution time of the operation for the caller. Otherwise, it will be aborted as a time fence error. The objective of this extension to a normal object paradigm is to prevent timing errors from crossing task or module boundaries (as often happens in traditional real-time systems which use a cyclic executive) and bind the timing error at every object invocation.

On top of the ARTS foundation we have built a relational database manager using message passing primitives and employing the client/server paradigm. The result, RTDB, currently consists of a multi-threaded server which accepts requests of several clients. Based on the temporal urgency of the request, the server determines whether it can commit the transaction or it has to reject it.

## 3. Comparison with Existing Systems

One of the principal goals of the ARTS project is to provide a more easily extensible real-time environment than is currently enjoyed by programmers developing on other kernels. To that end, ARTS requires better data management facilities than many other kernels offer. The RTDB on ARTS represents a combination of desirable aspects of database technology and development flexibility. In comparing the RTDB with other existing systems, we note some differences between it and both research and commercial products. For example, the CASE-DB is developed as a single-user, disk-based, real-time relational DBMS, which uses the relational algebra as its query language [Ozso90]. RTDB diverges from this design philosophy in many ways, being a multi-user, distributed real-time DBMS.

Supported media types also differ among real-time environments. The HP-RTDB, one of Hewlett Packard's Industrial Precision Tools, provides software application developers with a tool to structure and access memory-resident data [Fate89]. Essentially, HP-RTDB is a library of routines used to define and manipulate a database schema, build the database in memory, as well as load and unload, and write or read data to and from it. They also provide mechanisms for archiving schema and data, and storing timestamp information. The ARTS-RTDB provides a three tiered approach for supported media types, offering memory-resident data options, RAM-based disk storage, and access to the UNIX file system for disk storage. Each media has its own advantages and drawbacks in terms of compatibility, performance, and recoverability. Naturally, access times decrease along this continuum. This support of various media types provides developers the flexibility to choose appropriately those that best suit their needs. Also, we provide the ability to cross the boundaries between these media, and to utilize several media types in an individual query for both the source and resultant relations.

## 4. The RTDB Real-Time Database Manager

The RTDB is a relational database manager written in C designed to run on ARTS. It offers not only a functionally complete set of relational operators— such as join, projection, selection, union, and set difference— but also other necessary operators as create, insert, update, delete, rename, compress, sort, extract, import, export, and print. These operators give the user a good amount of relational power and convenience in managing the database.

We have developed two different kinds of clients for the RTDB. One is an interactive command parser/request generator that makes requests to the server on behalf of the user. This client looks and

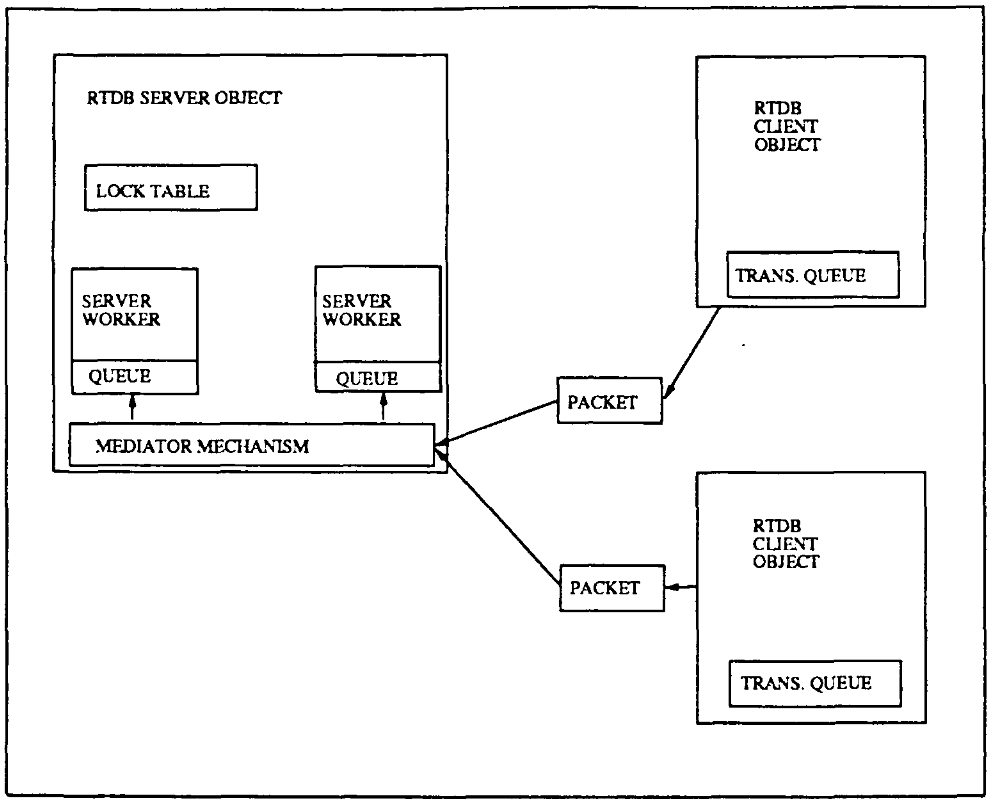


Figure 1. Mediator object model

behaves similarly to a single-user database manager. It is possible to run the client without knowing that any interaction between server and client is occurring. The other client is a transaction-generating "batch" client, representing a real-time process that needs to make database access requests.

The RTDB server object is the heart of the database management system. It is responsible for creating and storing the relations, receiving and acting on requests from multiple clients, and returning desired information to the clients.

The server object defines three threads. The root thread is automatically executed by ARTS upon invocation of the server. The server activates one or more worker threads, and activates a backup thread which is responsible for periodically backing up the relations that reside only in main memory.

The root thread of the server is responsible for binding the server's name in the ARTS name server so that the clients can find it and send requests. It is also responsible for reading the relations into memory, initializing the lock table, initializing the blocked request queue, instantiating the backup thread, and instantiating the server worker threads. There is at least one server worker created for each thread priority. After completing these tasks, the root thread enters an infinite loop that accepts database requests from any client. The requests come in as packets. The RTDB provides two different types of packets: *call* packets and *return* packets. The call packet, created by a client, contains all the information that the server needs to carry out the desired database access operation. Since different commands require different information, the call packet has a variant field containing different information for each command. When the server completes the processing of the request, it returns a packet to the client with the information requested. This packet is called a return packet. The return packet is created by the server and also has a variant field that carries command specific information.

The communication between the server and clients is performed by the ARTS communication primitives: Request, Accept, and Reply. The communication is synchronous; when a client issues a Request, it is blocked until the server Accepts and Replies to the message. This may cause some problems, especially in a real-time environment, for two reasons: priority inversion and data sharing.

The ARTS kernel (and thus the RTDB system) supports eight message priorities. When the root thread Accepts a message, it extracts priority information from the message packet. The root thread then enqueues the request on the message queue (i.e. pending request queue) of the worker thread designated to service requests of that priority level. If inactive, the server worker will be polling its queue; if active, the requests will be processed in FIFO order. Note that in this way we can easily exploit the scheduling merits of the underlying ARTS kernel without circumventing its priority-based scheduling mechanisms. Since the worker thread's priority matches that of the messages it services, it will only be scheduled for the CPU in an interval where its priority is currently the highest in the system. This is for a general case. For those instances where the scheduling technique is not priority based, or ARTS priority inheritance mechanism is employed, these decisions will naturally be reflected in the workers.

This technique of distributing requests among a pool of workers based on information contained in the request packet is called *workload mediation.* It is intrinsic in implementing various algorithms which deal with semantic information provided by the clients and/or the task requests (i.e. temporal issues, priorities, etc.). Determining the proper balance of control between ARTS primitives and RTDB explicit mediation will help us achieve the most beneficial symbiosis of the system's resources, which is one of the goals of our research. Figure 1 illustrates the mediator mechanism incorporated within the server object.

The worker thread of the RTDB server performs the client's request to access the database. It checks its request message queue, carries out the work that is requested, and replies back to the client. The worker Replies to a client without completing a request when it needs to return more information than can fit in a single packet. In such a case, the client must make continuation requests to the server until it gets all the information requested.

To maintain the consistency of the database, the RTDB server needs to handle conflicting requests properly. For example, a problem occurs when some request or part of a request (as in a multi-relational query) has to be blocked since it needs to lock a relation that is already locked. Our solution to this problem is to use a lock table that keeps track of which relations are in use at any given time. If a request for file *A* comes in while file *A* is being used by another active worker, then the new request must be put on an internal queue until *A* and any other files it needs are available.

Whenever the worker becomes free, it first checks its queue of blocked requests. If there are any requests in the block queue that can be unblocked, it dequeues the request and processes it. If no request in the block queue is ready to be processed, the worker looks to its incoming request queue.

## 5. Programming Interfaces

Conventional database systems often provide some interface through which they export functionality to application developers. Such programming interfaces simplify storage and retrieval tasks and provide a scheme for the creation, manipulation and destruction of database files. For systems utilizing the client-server paradigm, communication primitives can also be accessed through such an interface, achieving further hiding of the implementation details.

To facilitate the construction of application clients, we seek to provide a programming interface for the database command set which hides the implementation details of the system as much as possible. In this way, developers who are more familiar with function-call interfaces (e.g. SQL) can quickly adjust to the task of constructing custom application clients rather than application programs. Programming interfaces in real-time databases differ greatly in terms of application-developer friendliness. Some DBMS interfaces are tightly coupled to theoretical techniques such as the relational algebra. CASE-DB [Ozso90] is an example of this type of interface. While this interface satisfies the desired functionality requirements for a database, it can be awkward to use when developing large, complex applications. For these applications, it is more appropriate to use an interface similar to those already in use in non-real-time systems. These application program interfaces consist of library functions.

In designing the programming interface for RTDB, in addition to providing routines as in other relational databases, we seek to hide the details of ARTS' Request/Accept/Reply message passing sequence. This allows interaction between client and server to appear as if the application client were the only one interacting with the server. This goal is only partially attainable, since the physical code provided by the application developer must coexist in the same source code file as code which specifies constants and declarations necessary to construct the complete client image. To expedite the development process, we provide a thoroughly commented, standardized *client template* with which developers need only combine their source and compile.

## 6. DOSE: An Application of the RTDB

One of the applications of the RTDB is the Distributed Operating System Experiment (DOSE), as presented in [Butt90]. The goal of DOSE is to evaluate the feasibility of using a database kernel in embedded systems with requirements for high performance and real-time priority and predictability guarantees.

The DOSE application consists of data input, storage, display, and retrieval functions. These functions are implemented by four components: parser manager (PM), track report manager (TRM), graphics map client (GMC), and database monitor client (DMC). Figure 2 illustrates the information flow among objects in the DOSE experiment.

The PM receives tracking data from data terminals or communication links and converts them into a useful format such as floating point and signed integer numbers. The PM does not retain any incoming or outgoing information. The parsed data coming out from the PM are stored by the TRM. For each new incoming tracking data, a new data object is created. For high reliability, TRM maintains replicated data objects. The GMC enables the data to be mapped out and visualized on screen. It periodically checks with the TRM for the latest updates to be displayed. The DMC monitors the data objects in each replicated TRM database. Using frequent updates, it guarantees that data would remain consistent across the replicated TRM databases. Without DMC, the survivability and consistency of the system would be weakened.

The scenario used with the DOSE application is an outer air battle scenario generated by IBGTT, the Interim Battle Group Tactical Trainer. The data generated by IBGTT consists of coordinate and motion data as well as general military classifications of tracked objects, called *platforms*. This data can be used to plot tactical information for a variety of situations, including personnel training programs, strategic simulations, and real-time military surveillance. Table 1 shows the attributes of data objects used in the DOSE application.

Since some of the fields above are basically used as *categorical designators*, or flags, they can be used in simple boolean subqueries (e.g., "where clauses" in the RTDB syntax). For example, an "H" value for attribute *cat* indicates a hostile platform; an "F", a friendly platform. A "Y" value for attribute *nuclear* indicates a confirmed nuclear platform; a "N" value is a confirmed non-nuclear platform, and "U" is unknown. For example, a query which seeks to determine all the attributes of the friendly nuclear platforms may look as follows:

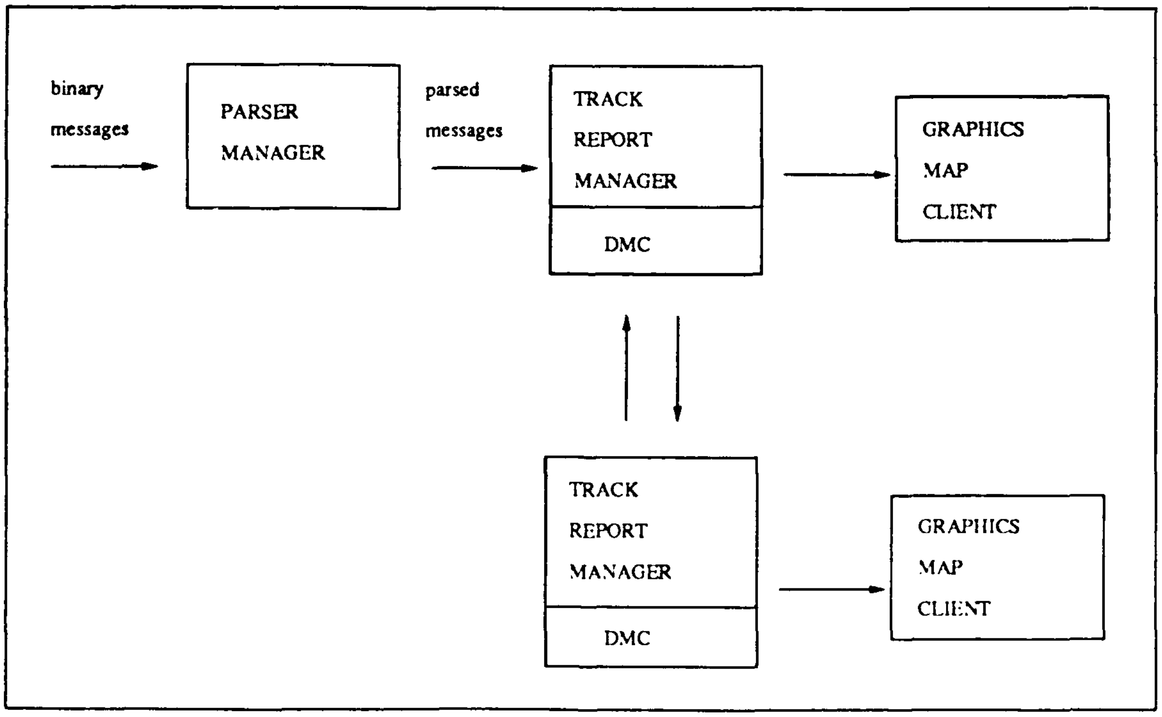


Figure 2. Information flow among objects in the DOSE experiment

| Attribute Name | Type | Meaning |
|---|---|---|
| trk_num | integer | track number |
| lat_track | real | latitude of track |
| long_track | real | longitude of track |
| bearing | real | bearing from data link reference point |
| dep_high | real | depth or height of platform |
| lat_dlrp | real | latitude of data link reference point |
| long_dlrp | real | longitude of data link reference point |
| platform_type | char | type of platform |
| cat | char | category of platform |
| time | integer | greenwich mean time |
| trkqa | integer | confidence of measurements |
| lat_tdir | char | latitude direction |
| long_tdir | char | longitude direction |
| course | real | bearing minus data link reference point |
| speed | real | speed of platform |
| range | real | range from data link reference point in nautical miles |
| nuclear | char | nuclear classification of platform |

Table 1. Data object attributes in the DOSE application

```
RTDB ==> print * from trackfile
where cat = "F" and nuclear = "Y";
```

A query to display information on all platforms in a certain track might look like this:

```
RTDB ==> print lattrk, longtrk, bearing,
    nuclear from trktable1 where trk = 4741;
```

In implementing the TRM and DMC using the RTDB, the original DOSE tracking data has been decomposed into several track files of similar data. All commands currently supported by the RTDB have been tested on the trackfile data.

Modifications to earlier versions of RTDB have been made to support attribute type REAL which is identical to the floating point type of DOSE application, and to support aggregate operators such as SUM, COUNT, MIN, MAX, and AVG. Although not specifically delineated in the queries proposed in the DOSE application, the addition of aggregates seems important for the type of queries expected for the application. For example, the system may be called to report a COUNT of all hostile, nuclear, air platforms, or the bearing and speed of the hostile platform with MIN range.

Temporal database components are being investigated for inclusion in the RTDB for DOSE application. They will address the desired timestamping of surveillance updates generated by radar, sonar, or similar equipments, and temporal consistency requirements of real-time transactions. Other potential improvements in efficient implementation are being examined to determine their overall value to the RTDB system. Indices and views are two of them. Since such features not only alter the speed and predictability of the system but also the basic file structure, they need to be examined closely on their own, and then as new elements within the existing system.

## 7. Conclusions

A real-time database manager is one of the critical components of real-time systems, in which tasks are associated with deadlines and a significant portion of data is highly perishable in the sense that it has value to the mission only if used quickly. To satisfy the timing requirements, transactions must be scheduled considering not only the consistency constraints but also their timing constraints. In addition, the system should support a predictable behavior such that the possibility of missing deadlines of critical tasks could be informed ahead of time, before their deadlines expire.

In this paper, we have presented an experimental relational database manager developed for distributed real-time systems. Since the characteristics of a real-time database manager are distinct from conventional database managers, there are different kinds of issues to be considered in developing a real-time database manager. For example, priority-based scheduling and memory resident data have been investigated in the development of the RTDB.

The foundation now exists for a real-time relational database manager. However, as with any active research project, there are many technical issues associated with real-time database systems that need further investigation. It is our goal to facilitate further development in this area, as well as with our RTDB. To that end, we have discussed our work toward providing a flexible programming interface and standard client template to allow quick prototyping and faster modeling. The RTDB described in this paper with its multi-threaded server model, is an appropriate research vehicle for investigating new techniques and scheduling algorithms for distributed real-time database systems.

## References

[Abb89] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *VLDB Conference*, August 1989.

[Buc89] Buchmann, A. et al., "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control," *Fifth Data Engineering Conference*, Feb. 1989, 470-480.

[Butt90] Butterbrodt, M. and J. Green, "DOSE: A Vital Link in the Development of a Real-Time Relational Database Environment," *Project Summary*, Naval Ocean Systems Center, Jan. 1990.

[Fate89] Fatehi, Feyzi, "With the Speed of the Winged Horse: Hewlett-Packard Real-Time Database," *Internal Document*. Hewlett Packard Company, Apr. 1989.

[IEEE90] *Seventh IEEE Workshop on Real-Time Operating Systems and Software*, University of Virginia, Charlottesville, Virginia, May 1990.

[Kor90] Korth, H., "Triggered Real-Time Databases with Consistency Constraints," *16th VLDB Conference*, Brisbane, Australia, Aug. 1990.

[Lin90] Lin, Y. and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *11th IEEE Real-Time Systems Symposium*, Orlando, Florida, Dec. 1990, to appear.

[ONR90] *ONR Annual Workshop on Foundations of Real-Time Computing*, Washington, DC, Oct. 1990.

[Ozso90] Ozsoyoglu, G., et al., "CASE-DB--A Real-Time Database Management System," *Tech. Rep.* Case Western Reserve University, 1990.

[Sha88] Sha, L., R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988, 82-98.

[Sha91] Sha, L., R. Rajkumar, S. H. Son, and C. Chang, "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, to appear.

[Son88] Son, S. H., "Real-Time Database Systems: Issues and Approaches," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988.

[Son89] Son, S. H. and H. Kang, "Approaches to Design of Real-Time Database Systems," *International Symposium on Database Systems for Advanced Applications*, Seoul, Korea, April 1989, 274-281.

[Son90] Son, S. H. and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," *10th International Conference on Distributed Computing Systems*, Paris, France, June 1990, 124-131.

[Son90b] Son, S. H. and J. Lee, "Scheduling Real-Time Transactions in Distributed Database Systems," *7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, Virginia, May 1990, 39-43.

[Son91] Son, S. H., P. Wagle, and S. Park. "Real-Time Database Scheduling: Design, Implementation, and Performance Evaluation," *International Symposium on Database Systems for Advanced Applications* (DASFAA '91), Tokyo, Japan, April 1991.

[Stan88] Stankovic, J., "Misconceptions about Real-Time Computing," *IEEE Computer 21*, 10, October 1988, 10-19.

[Ston81] Stonebraker, M., Operating System Support for Database Management, *Commun. of ACM 24*, 7 (July 1981), 412-418.

[Tok89] Tokuda, H. and C. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, 23 (3), July 1989.

# An Environment for Integrated Development and Evaluation of Real-Time Distributed Database Systems

Sang H. Son
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

## ABSTRACT

Real-time database systems must maintain consistency while minimizing the number of transactions that miss the deadline. To satisfy both the consistency and real-time constraints, there is the need to integrate synchronization protocols with real-time priority scheduling protocols. One of the reasons for the difficulty in developing and evaluating database synchronization techniques is that it takes a long time to develop a system, and evaluation is complicated because it involves a large number of system parameters that may change dynamically. This paper describes an environment for investigating distributed real-time database systems. The environment is based on concurrent programming kernel which supports the creation, blocking, and termination of processes, as well as scheduling and inter-process communication. The contribution of the paper is the introduction of a new approach to system development that utilizes a module library of reusable components to satisfy three major goals: modularity, flexibility, and extensibility. In addition, experiments of real-time concurrency control techniques are presented to illustrate the effectiveness of the environment.

Index Terms - distributed database, prototyping, synchronization, transaction, real-time

---

This work was supported in part by ONR contract # N00014-91-J-1102.

## 1. Introduction

In this paper, we report our experiences with a new approach to integrated development and evaluation of real-time distributed database systems, and present experimental results of various real-time synchronization techniques. The goal of the project is to test the hypothesis that a host environment can be used to significantly accelerate the rate at which we can perform experiments in the areas of operating systems, databases, and network protocols for real-time systems. A tool for developing components of real-time distributed systems and integrating them to evaluate design alternatives is essential for the advance of real-time computing technology. To the best of our knowledge, this is the first successful attempt to develop such a tool as an environment consisting of a hybrid of actual implementation and simulation.

As computers are becoming essential part of real-time systems, *real-time computing* is emerging as an important discipline in computer science and engineering [Shin87]. The growing importance of real-time computing in a large number of applications, such as aerospace and defense systems, industrial automation, and nuclear reactor control, has resulted in an increased research effort in this area. Researchers working on developing real-time systems based on distributed system architecture have found out that database managers are assuming much greater importance in real-time systems. In the recent workshops developers of "real" real-time systems pointed to the need for basic research in database systems that satisfy timing constraint requirements in collecting, updating, and retrieving shared data [IEEE90, ONR90]. Further evidence of its importance is the recent growth of research in this field and the announcements by some vendors of database products that include features achieving high availability and predictability [Son88].

In addition to providing relational access capabilities, distributed real-time database systems offer a means of loosely coupling software processes; therefore, making it easier to rapidly update software, at least from a functional perspective. However, with respect to time-driven scheduling and system timing predictability, they present new problems. One of the characteristics of current database managers is that they do not schedule their transactions to meet response requirements and they commonly lock data tables indiscriminately to assure database consistency. Locks and time-driven scheduling are basically incompatible. Low priority transactions can and will block higher priority transactions leading to response requirement failures. New techniques are required to manage database consistency which are compatible with time-driven scheduling and the essential system response predictability/analyzability it brings. One of the primary reasons for the difficulty in successfully developing and evaluating new database techniques is that it takes a long time to develop a system, and evaluation is complicated because it involves a large number of system parameters that may change dynamically.

A prototyping technique can be applied effectively to the evaluation of database techniques for distributed real-time systems. In this paper, we report our experiences with a new database prototyping environment. It is constructed to support research in distributed database and operating system technology for real-time applications. A *database prototyping environment* is a software package that supports the investigation of the properties of database techniques in an environment other than that of the target database system. The advantages of an environment that provides prototyping capability are obvious. First, it is cost effective. If experiments for a twenty-node distributed database system can be executed in a software environment, it is not necessary to purchase a twenty-node distributed system, reducing the cost of evaluating design alternatives. Second, design alternatives can be evaluated in a uniform environment with the same system parameters, making a fair comparison. Finally, as technology changes, the environment need only be updated to provide researchers with the ability to perform new experiments.

A prototyping environment can reduce the time of evaluating new technologies and design alternatives. From our past experience, we assume that a relatively small portion of a typical database system's code is affected by changes in specific control mechanisms, while the majority of code deals with intrinsic problems, such as file management. Thus, by properly isolating technology-dependent portions of a database system using modular programming techniques, we can implement and evaluate design alternatives very rapidly. In addition, a prototyping environment provides a friendlier development environment than a target hardware system. The bare machine environment is the worst possible place in which to explore new software concepts. For example, even the recovery of the event history leading up to an error in a distributed system can be a difficult and, in some cases, an impossible task. Debugging is greatly facilitated in a prototyping environment. The symbolic debugger of our environment supports the examination of an arbitrary number of execution threads. As a result, the state of a distributed computation can be examined as a whole.

Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation, especially for distributed real-time database systems. Recently, simulators have been developed for investigating performance of several concurrency control algorithms for real-time applications [Abb88, Abb89, Raj89]. However, they do not provide a module hierarchy composed from reusable components as in our prototyping environment. Software developed in our prototyping environment will execute in a given target machine without modification of any layer except the hardware interface. In addition, since our environment is a hybrid of prototyping and simulation (i.e., partially implemented and partially simulated), we can capture important timing features of the system, while it is very hard using simulation only.

A database system must operate in the context of available operating system services. In other words, database operations need to be coherent with the operating system, because correct functioning and timing behavior of database control algorithms depend on the services of the underlying operating system. Unless you have a control over the operating system, investigating timing behavior of a database system does not provide much information. An environment for database systems development must, therefore, provide facilities to support operating system functions and integrate them with database systems for experimentation.

Another important use of a prototyping environment is to analyze the reliability of database control mechanisms and techniques. Since distributed systems are expected to work correctly under various failure situations, the behavior of distributed database systems in degraded circumstances needs to be well understood. Although new approaches for synchronization and checkpointing for distributed databases have been developed recently [Liu87, Kor90, Lin90, Son89, Son90], experimentation to verify their properties and to evaluate their performance has not been performed due to the lack of appropriate test tools.

When developing a database system, functional completeness and performance of the system are of primary concern. The resulting systems are often not layered or modular in their implementation. However, for experimentation, a layered implementation approach facilitates the rapid evaluation of new techniques. Such a facility improves significantly the capability of the system designer in comparing design alternatives in a uniform environment. In this regard, the concept of developing a methodology for layered implementation of the system and building a library of modules with different performance/reliability characteristics for operating system and database system functions seems promising. The prototyping environment we have developed follows this approach [Cook87, Son88b].

The rest of the paper is organized as follows. Section 2 presents an informal description of a message-based simulation. Section 3 describes the design principles and the current implementation of the prototyping environment. Section 4 presents experimentations of priority-based synchronization algorithms and multiversion data objects using the prototyping environment. Section 5 concludes the paper.

## 2. Message-Based Simulation

When prototyping distributed database systems, there are two possible approaches: sequential programming and distributed programming based on message-passing. Message-based simulations, in which events are message-communications, do not provide additional expressive power over standard simulation languages; message-passing can be simulated in many discrete-event simulation languages including SIMSCRIPT [Kiv69] and GPSS [Sch74]. However, a message-based simulation can be used as an effective tool for developing a distributed system because the simulation "looks" like a distributed program, while a simulation program written in a traditional simulation language is inherently a sequential program. Furthermore, if a simulation program is developed in a systematic way such that the principles of modularity and information hiding are observed, most of the simulation code can be used in the actual system, resulting in a reduced cost for system development and evaluation.

To prototype a distributed database system on a single host machine, it is necessary to provide virtual machines for each node of the system being simulated. For that, the process view of a system has been adopted. A distributed system being simulated consists of a number of *processes* which interact with others at discrete instants of time. Processes are basic building blocks of a simulation program. A process is an independent, dynamic entity which manipulate *resources* to achieve its objectives. A resource is a passive object and may be represented by a simple variable or a complex data structure. A simulation program models the dynamic behavior of processes, resources, and their interactions as they evolve in time. Each physical operation of the system is simulated by a process, and the process interactions are called *events*.

In the literature, the notion of a process has been given numerous definitions. The definition used in our model is much the same as that given in [Bri78]: A process is the execution of an interruptible sequential program and represents the unit of resource allocation, such as the allocation of CPU time, main memory and I/O devices.

We use the client/server paradigm for process interaction in the prototyping environment. The system consists of a set of clients and servers, which are processes that cooperate for the purpose of transaction processing. Each server provides a service to its clients, where a client can request a service by sending a request message (a message of type *request)* to the corresponding server. The computation structure of the system to be modeled can be characterized by the way clients and servers are mapped into processes. For example, a server might consist of a fixed number of processes, each of which may execute requests from every transaction, or it might consists of varying number of processes, each of which executes on behalf of exactly one transaction.

Internal actions of a process, i.e., actions that do not involve interactions with other processes in the system, are modeled either by the passage of simulation time or by the execution of sequential statements within the process. We use a simulator clock to represent the passage of time in a simulation. The simulator clock advances in discrete steps, where each step simulates the passage of time between two events in the system.

In a physical system, each process makes independent progress in time ` the resources they need are available, and many processes execute in parallel. In its simulation, the multiple processes of a physical system must be executed simultaneously on one processor. This simultaneity is achieved in the prototyping environment by supporting a simultaneous execution of multiple processes in a single address space.

A message-based prototyping environment can be of enormous benefit in designing and testing emerging systems, such as real-time systems, and in comparing and improving algorithms that are applicable to many different systems. One such benefit is that the software to be used in an actual system can be developed using the environment. The prototyping environment can support a simulated environment, actual hardware, or a "hybrid" mode in which some of the modules are implemented in hardware and some are simulated. In this way, it is irrelevant to the software developer using the environment whether or not all or part of the software is running on hardware. When the system is running in a hybrid mode, the virtual clock used for performance measurement is updated by the actual time used for direct execution, making performance measurements correct.

## 3. Structure of the Prototyping Environment

The prototyping environment is designed to facilitate easy extensions and modifications. Server processes can be created, relocated, and new implementations of server processes can be dynamically substituted. It efficiently supports a spectrum of real-time database functions at the operating system level, and facilitates the construction of multiple database systems with different characteristics. For experimentation, system functionality can be adjusted according to application-dependent requirements without much overhead for new system setup. Since one of the design goals of the prototyping environment is to conduct an empirical evaluation of the design and implementation of real-time distributed database systems, it has built-in support for performance measurement of both elapsed time and blocked time for each transaction.

The prototyping environment provides support for transaction processing, including transparency to concurrent access, data distribution, and atomicity. An instance of the prototyping environment can manage any number of virtual sites specified by the user. Modules that implement transaction processing are decomposed into several server processes, and they communicate among themselves through ports. The clean interface between server processes simplifies incorporating new algorithms and facilities into the prototyping environment, or testing alternate implementations of algorithms. To permit concurrent transactions on a single site, there is a separate process for each transaction that coordinates with other server processes.

Figure 1 illustrates the structure of the prototyping environment. The prototyping environment is based on a concurrent programming kernel, called the StarLite kernel. The StarLite kernel supports process control to create, ready, block, and terminate processes. It also supports the semaphore abstraction to be used by higher level modules in resource control, critical section implementation, and synchronous message-passing. The internal structure of the kernel follows the well-known *client-server model* [Tan87], in which most of the operating system operates as server processes in the same address space as client processes, with the kernel merely handling message communication between various processes. Figure 2 shows an instance of this model. This structure is particularly useful for extensible systems such as our prototyping environment, since additional or alternative functionality can easily be provided by creating a new server, instead of changing and recompiling the kernel.

Scheduler in the kernel maintains a virtual clock and provides the **hold** primitive to control the passage of time. The benefit of a virtual clock is that any number of performance monitoring operations may be performed at an instant of virtual time. If a physical clock were embedded, the monitoring activities themselves would interfere with other system activities and add to the execution time, resulting in incorrect performance measures.

The kernel also provides the capability of isolating overhead imposed by each system component. For instance, total time at each node can be divided into CPU time and I/O time, to determine the computation-intensive and I/O-intensive functions and investigate the distribution of tasks around the system so as to maximize parallelism.

The User Interface (UI) is a front-end invoked when the prototyping environment begins. UI is menu-driven, and designed to be flexible in allowing users to experiment various configurations with different system parameters. A user can specify the following:

- system configuration: number of sites and the number of server processes at each site, topology and communication costs.
- database configuration: database at each site with user defined structure, size, granularity, and levels of replication.

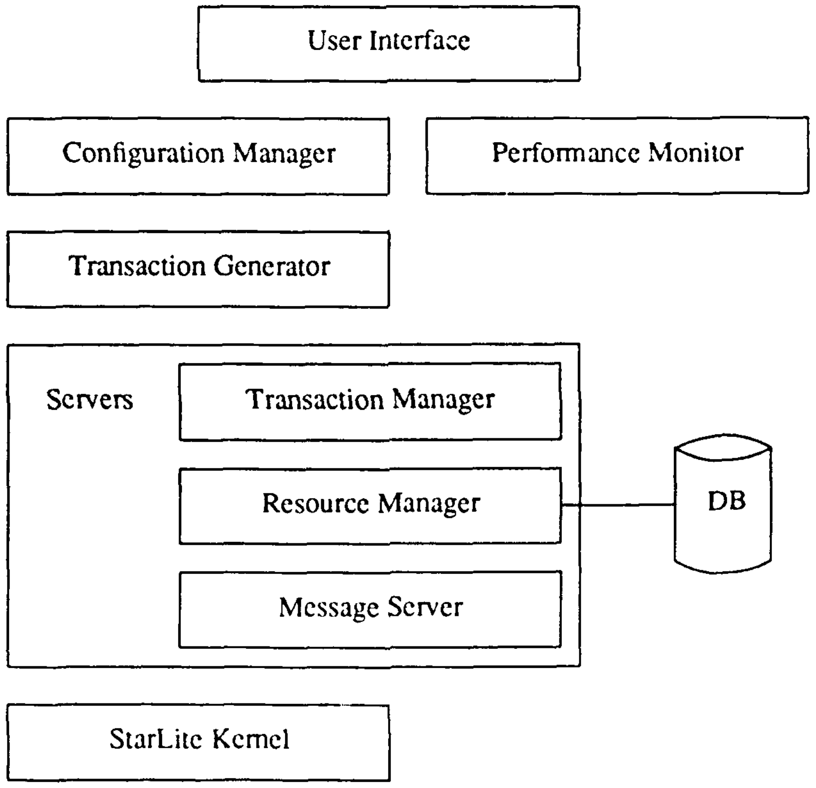


Fig. 1. Structure of the prototyping environment

- load characteristics: number of transactions to be executed, size of their read-sets and write-sets, transaction types (read-only or update) and their priorities, and the mean interarrival time of transactions.
- concurrency control: locking, timestamp ordering, and priority-based.

The UI initiates the Configuration Manager (CM) which initializes necessary data structures for transaction processing based on user specification. The database at each site consists of different number of files, and each file consists of different number of records. The database structure can be made complicated if necessary. However, we use a simple file access, since investigating synchronization problems does not require complex database structures.

The CM invokes the Transaction Generator at an appropriate time interval to generate the next transaction to form a Poisson process of transaction arrival. The environment is flexible enough to generate any number of transactions with different characteristics. The user can specify his own procedure for transactions. At initialization time, the user-specified procedure is converted into a transaction process. Furthermore, the prototyping environment supports the facility that allows mixing system generated transactions with user-specified ones. It is very desirable to have such a capability, since the user can setup any workload that represents the situation to be simulated, with or without system generated background workload.

A transaction is distinguished from the other processes in the system by its behavior. To the system, the only distinction between transactions and server processes is the PortTags on which each receives messages. When a transaction is generated, it is assigned an identifier that is unique among all transactions in the system. Each transaction is also assigned a globally unique timestamp which is hidden within a single module. The advantage of extracting the definition and assignment of the timestamp from its use is that it provides a means of uniquely assigning timestamps which is independent from any specific implementation.

The timestamp assignment is closely related to the clocks in the system. In a sequential simulation, a single clock suffices to order events in the system. An event is taken off the event queue, and the global

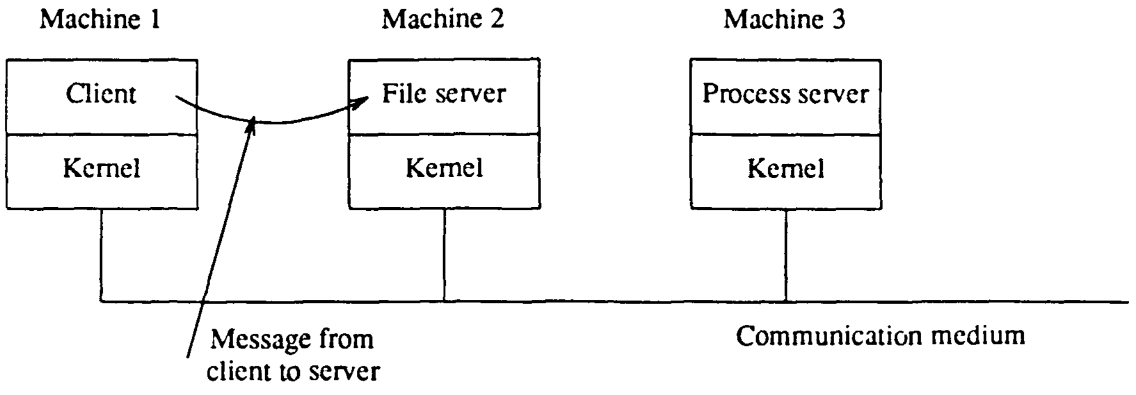


Fig. 2. Client-server model

clock is advanced to the time required for the event to occur. Events are related in time by their relation to the global clock. In prototyping distributed environments, no such global clock is available. Time is referred to by local clocks, which is maintained at each site and visible only to processes at that site. Ordering of events in terms of the global time, therefore, depends on the proper synchronization of local clocks. In our environment, clocks are synchronized by intersite communication. An intersite message includes the clock value of the sender site at the time the message is sent. If the sum of this clock value and the propagation delay between the sites is greater than the clock value at the receiver site, the receiver increments its clock by the difference between the sum and its clock value. In this way, all succeeding events at the receiver site can be said to occur *after* the sending of the message. This satisfies our intuitive notion of "happens before" relationship [Lam78].

Transaction execution consists of read and write operations. Each read or write operation is preceded by an access request sent to the Resource Manager, which maintains the local database at each site. Each transaction is assigned to the Transaction Manager (TM). The TM issues service requests on behalf of the transaction and reacts appropriately to the request replies. For instance, if a transaction requests access to a file and that file is locked, TM executes either blocking operation to wait until the data object can be accessed, or aborting procedure, depending on the situation. If granting access to a resource will produce deadlock, TM receives an abort response and aborts the transaction. Transactions commit in two phases. The first commit phase consists of at least one round of messages to determine if the transaction can be globally committed. Additional rounds may be used to handle potential failures. The second commit phase causes the data objects to be written to the database for successful transactions. TM executes the two commit phases to ensure that a transaction commits or aborts globally. Figure 3 illustrates a queueing model adopted for transaction processing.

Transactions are generated and put into the start-up queue. When a transaction is started, it leaves the start-up queue and enters the ready queue. The transaction at the top of the queue is selected to run. The current running transaction sends requests to the concurrency controller (CC) implemented in the resource manager. The transaction may be blocked and placed in the block queue. It may also be aborted and restarted. In such a case, it is first delayed for a certain amount of time and then put in the ready queue again. When a transaction in the block queue is unblocked, it leaves the block queue and is placed in the ready queue again.

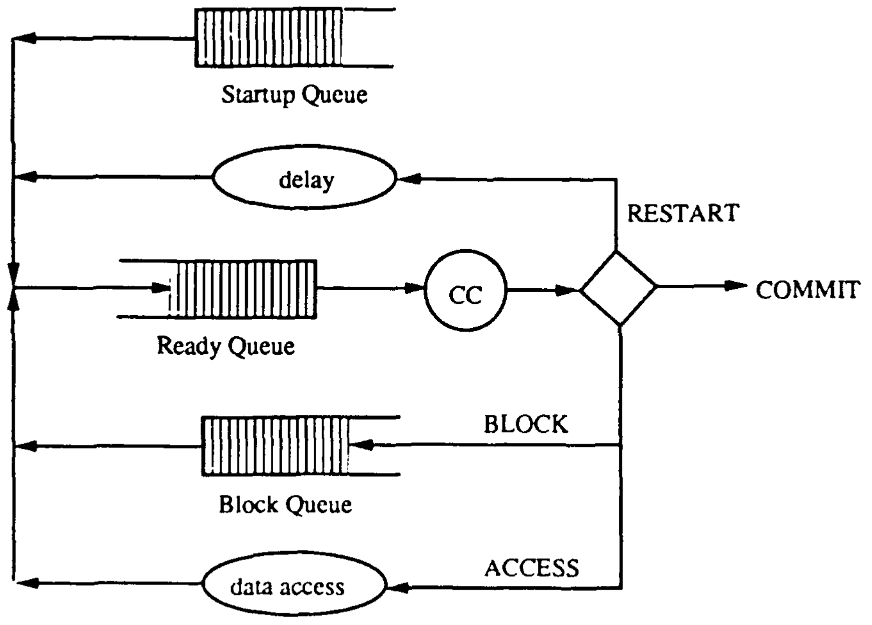


Fig. 3 Simulation Model

In prototyping distributed database systems, a communication network is an important component to be simulated, since the system performance depends heavily on the topology and communication protocols used. However, in many database simulators, the communication subsystem is either ignored or simplified by adding communication cost to the transaction processing time. Our prototyping environment uses a different approach by providing a virtual communication network that actually runs a layered communication protocol on a network topology specified by the user. Since the communication module is a separate building block in the prototyping environment, the user can change it to simulate different requirements of the application.

The Message Server (MS) is a process listening on a well-known port for messages from remote sites. When a message is sent to a remote site, it is placed on the message queue of the destination site and the sender blocks itself on a private semaphore until the message is retrieved by MS. If the receiving site is not operational, a time-out mechanism will unblock the sender process. When MS retrieves a message, it wakes the sender process and forwards the message to the proper servers or TM. The prototyping environment supports both Ada-style rendezvous (synchronous) as well as asynchronous message passing. Inter-process communication within a site does not go through the Message Server; processes send and receive messages directly through their associated ports

The inter-process communication structure is designed to provide a simple and flexible interface to the client processes of the application software, independent from the low-level hardware configurations. It is split into three levels of hierarchy: transport, network, and physical layers.

The Transport layer is the interface to the application software, thus it is designed to be as abstract as possible in order to support different port structures and various message types. In addition, application level processes need not know the details of the destination device. The invariant built into the design of the inter-process communication interface is that the application level sender allocates the space for a message, and the receiver deallocates it. Thus, it is irrelevant whether or not the sender and receiver share memory space, i.e., whether or not the Physical layer on the sender's side copies the message into a buffer and deallocates it at the sender's site, and the Physical layer at the receiver's site allocates space for the message. This enables prototyping distributed systems or multiprocessors with no shared memory, as well as multiprocessors with shared memory space. When prototyping the latter, only addresses need to be passed in messages without intermediate allocation and deallocation.

The Physical layer of message passing simulates the physical sending and receiving of bits over a communication medium, i.e., it is for intersite message passing. The device number in the interface is simply a cardinal number; this enables the implementation to be simple and extensible enough to support any application. To simulate sending or to actually send over an Ethernet in the target system, for example, a module could map network addresses onto cardinals. To send from one processor to another in a multiprocessor or in a distributed system, the cardinals can represent processor numbers.

Messages are passed to specific processes at specific sites in the Network layer of the communications interface. This layer serves to separate the Transport and the Physical layers, so that the Transport layer interface can be processor- and process-independent and the Physical layer interface need be concerned only with the sending of bits from one site to another. The Transport layer interface of the communication subsystem is implemented in the Transport module. A Transport-level Send is made to an abstraction called a *PortTag*. This abstraction is advantageous because the implementation (i.e., what a PortTag represents) is hidden in the Ports module. Thus the PortTag can be mapped onto any port structure or the reception points of any other message passing system. The Transport-level Send operation builds a packet consisting of the sender's PortTag, used for replies, the destination PortTag, and the address of the message. It then retrieves from the destination PortTag the destination device number. If this number is the same as the sender's, the Send is an intra-site message communication, and hence the Network-level Send is performed. Otherwise the send requires the Physical module for intersite communication.

Note that accesses to the implementation details of the PortTag are restricted to the module that actually implements it; this enables changing the implementation without recompiling the rest of the system.

The Performance Monitor interacts with the transaction managers to record, priority/timestamp and read/write data set for each transaction, time when each event occurred, statistics for each transaction and cpu hold interval in each node. The statistics for a transaction includes arrival time, start time, total processing time, blocked interval, whether deadline was missed or not, and the number of aborts.

Since each TM is a separate process, each has its own data area in which to keep track of the time when a service request is sent out and the time the response arrives, as well as the time when a transaction begins blocking, waiting for a resource, and the time the resource is granted. When a transaction commits, it calls a procedure that records the above measures; when the simulation clock has expired, these measures are printed out for all transactions.

## 4. Prototyping Real-Time Database Systems

The previous section described the structure of the prototyping environment, with some of its advanced features. In this section, we present real-time database systems implemented using the prototyping environment. The objectives of our study using the prototyping environment are 1) to evaluate the prototyping environment itself in terms of correctness, functionality, and modularity, and 2) performance comparison between two-phase locking and priority-based synchronization algorithms, and between a multiversion database and its corresponding single-version database, through the sensitivity study of key parameters that affect performance.

Compared with traditional databases, real-time database systems have a distinct feature: they must satisfy the timing constraints associated with transactions. In other words, "time" is one of the key factors to be considered in real-time database systems. The timing constraints of a transaction typically include its ready time and deadline, as well as temporal consistency of the data accessed by it. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on a tracking data of a missile must be processed within the given deadlines: otherwise, the information provided could be of little value. In such a system, transaction processing must satisfy not only the database consistency constraints but also the timing constraints.

The prototyping environment we have developed is especially useful for investigating timing behavior of real-time transactions, since we can control all the system components. An alternative to the prototyping approach is to develop a system on a bare machine, based on a specialized real-time kernel. The ARTS [Tok89] and the RT-CARAT [Hua90] take this approach. Difficulties with such an approach are 1) it takes much more effort to develop, 2) the system is strongly coupled with its hardware and hence hard to change its timing characteristics when needed, and 3) the system is not portable since it is implemented in the target environment.

In addition to providing shared data access capabilities, distributed real-time database systems offer a means of loosely coupling communicating processes, making it easier to rapidly update software, at least from a functional perspective. However, with respect to time-driven scheduling and system timing predictability, they present new problems. One of the characteristics of current database managers is that they do not schedule their transactions to meet response time requirements and they commonly lock data tables to assure database consistency. Locks and time-driven scheduling are basically incompatible. Low priority transactions holding locks required by higher priority transactions can and will block the higher priority transactions, leading to response requirement failures. New techniques are required to manage data consistency which are compatible with time-driven scheduling.

## 4.1. Steady State Estimation

In order to show that the results we get from experiments represent the performance of the system in steady states, we have performed experiments to check if the system were allowed to run for any length of time greater than certain threshold value, the variation in results would be within some tolerable interval. We have implemented a well-known synchronization protocol, two-phase locking (2PL), for the following system and workload configuration:

8 sites with fully interconnected network
multiprogramming level of 10

75% read-only and 25% update transactions
 read-only transactions access 3% of the database
 update transactions access 1% of the database
 database consists of 500 unreplicated objects
 Poisson distribution of transaction arrivals

Figure 4 shows the average response time of transactions using the 2PL. It shows that the average response time begins to stabilize at 3000 simulation time units, and varies only slightly from then on. The lower response time up to 3000 time units are due to the first set of transactions that benefits from a lower initial multiprogramming level and potential conflicts. In addition, since transactions requiring longer execution time will increase the average response time when they complete, they do not contribute to the average response time during the early stage of transaction execution if they were in the initial group of transaction. These initial characteristics are gradually erased from the average performance.

In addition, as we increase the time for experiments, the average response time is determined from an increasing number of transactions. For example, at 100 time units, the number of transactions contributing to the mean is approximately 12. At 4000, it is approximately 60. Thus the overall behavior of the system becomes less and less subject to the behavior of individual transactions. From the graph and characteristics of our environment, we concluded that an experiment must run at least 3500 time units

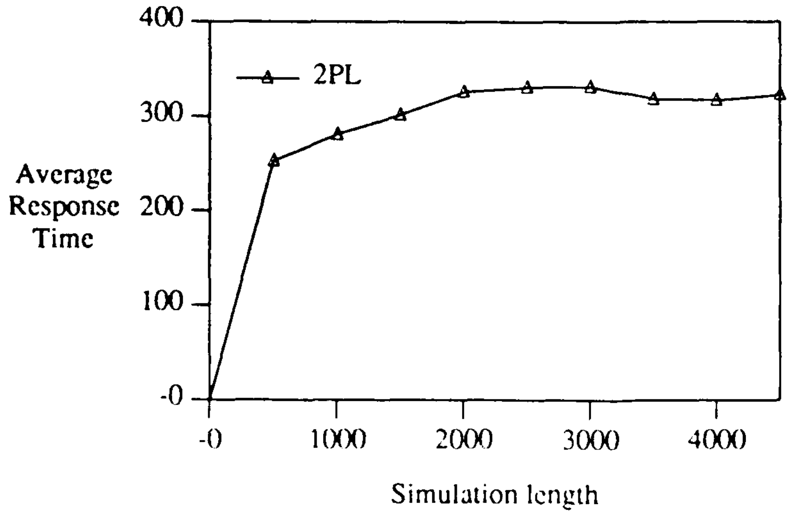


Fig. 4. Response time stability

before it starts to capture the steady state behavior of the system.

## 4.2. Priority-Based Synchronization

Real-time databases are often used by applications such as tracking. Tasks in such applications consist of both computing (signal processing) and database accessing (transactions). A task can have multiple transactions, which consists of a sequence of read and write operations operating on the database. Each transaction will follow the two-phase locking protocol, which requires a transaction to acquire all the locks before it releases any lock. Once a transaction releases a lock, it cannot acquire any new lock. A high priority task will preempt the execution of lower priority tasks unless it is blocked by the locking protocol at the database.

In a real-time database system, synchronization protocols must not only maintain the consistency constraints of the database but also satisfy the timing requirements of the transactions accessing the database. To satisfy both the consistency and real-time constraints, there is the need to integrate synchronization protocols with real-time priority scheduling protocols. A major source of problems in integrating the two protocols is the lack of coordination in the development of synchronization protocols and real-time priority scheduling protocols. Due to the effect of blocking in lock-based synchronization protocols, a direct application of a real-time scheduling algorithm to transactions may result in a condition known as *priority inversion* [Raj89]. Priority inversion is said to occur when a higher priority process is forced to wait for the execution of a lower priority process for an indefinite period of time. When the transactions of two processes attempt to access the same data object, the access must be serialized to maintain consistency. If the transaction of the higher priority process gains access first, then the proper priority order is maintained; however, if the transaction of the lower priority gains access first and then the higher priority transaction requests access to the data object, this higher priority process will be blocked until the lower priority transaction completes its access to the data object. Priority inversion is inevitable in transaction systems. However, to achieve a high degree of schedulability in real-time applications, priority inversion must be minimized. This is illustrated by the following example.

*Example:* Suppose $T_1$, $T_2$, and $T_3$ are three transactions arranged in descending order of priority with $T_1$ having the highest priority. Assume that $T_1$ and $T_3$ access the same data object $O_i$. Suppose that at time $t_1$ transaction $T_3$ obtains a lock on $O_i$. During the execution of $T_3$, the high priority transaction $T_1$ arrives, preempts $T_3$ and later attempts to access the object $O_i$. Transaction $T_1$ will be blocked, since $O_i$ is already locked. We would expect that $T_1$, being the highest priority transaction, will be blocked no longer than the time for transaction $T_3$ to complete and unlock $O_i$. However, the duration of blocking may, in fact, be unpredictable. This is because transaction $T_3$ can be blocked by the intermediate priority transaction $T_2$ that does not need to access $O_i$. The blocking of $T_3$, and hence that of $T_1$, will continue until $T_2$ and any other pending intermediate priority level transactions are completed.

The blocking duration in the example above can be arbitrarily long. This situation can be partially remedied if transactions are not allowed to be preempted; however, this solution is only appropriate for very short transactions, because it creates unnecessary blocking. For instance, once a long low priority transaction starts execution, a high priority transaction not requiring access to the same set of data objects may be needlessly blocked.

An approach to this problem, based on the notion of *priority inheritance*, has been proposed [Sha87]. The basic idea of priority inheritance is that when a transaction T of a process blocks higher priority processes, it executes at the highest priority of all the transactions blocked by T. This simple idea of priority inheritance reduces the blocking time of a higher priority transaction. However, this is inadequate because the blocking duration for a transaction, though bounded, can still be substantial due to the potential *chain of blocking*. For instance, suppose that transaction $T_1$ needs to sequentially access objects $O_1$ and $O_2$. Also suppose that $T_2$ preempts $T_3$ which has already locked $O_2$. Then, $T_2$ locks $O_1$. Transaction $T_1$ arrives at this instant and finds that the objects $O_1$ and $O_2$ have been respectively locked by the lower priority transactions $T_2$ and $T_3$. As a result, $T_1$ would be blocked for the duration of two transactions, once to wait for $T_2$ to release $O_1$ and again to wait for $T_3$ to release $O_2$. Thus a chain of blocking can be formed.

One idea for dealing with this inadequacy is to use a total priority ordering of active transactions [Sha88]. A transaction is said to be *active* if it has started but not yet completed its execution. A transaction can be active in one of two states: executing or being preempted in the middle of its execution. The idea of total priority ordering is that the real-time locking protocol ensures that each active transaction is executed at some priority level, taking priority inheritance and read/write semantics into consideration.

## 4.3. Total Ordering by Priority Ceiling

To ensure the total priority ordering of active transactions, three priority ceilings are defined for each data object in the database: the write-priority ceiling, the absolute-priority ceiling, and the rw-priority ceiling. The write-priority ceiling of a data object is defined as the priority of the highest priority transaction that may write into this object, and absolute-priority ceiling is defined as the priority of the highest priority transaction that may read or write the data object. The rw-priority ceiling is set dynamically. When a data object is write-locked, the rw-priority ceiling of this data object is defined to be equal to the absolute priority ceiling. When it is read-locked, the rw-priority ceiling of this data object is defined to be equal to the write-priority ceiling.

The priority ceiling protocol is premised on systems with a fixed priority scheme. The protocol consists of two mechanisms: *priority inheritance* and *priority ceiling*. With the combination of these two mechanisms, we get the properties of freedom from deadlock and a worst case blocking of at most a single lower priority transaction.

When a transaction attempts to lock a data object, the transaction's priority is compared with the highest rw-priority ceiling of all data objects currently locked by other transactions. If the priority of the transaction is not higher than the rw-priority ceiling, the access request will be denied, and the transaction will be blocked. In this case, the transaction is said to be blocked by the transaction which holds the lock on the data object of the highest rw-priority ceiling. Otherwise, it is granted the lock. In the denied case, the priority inheritance is performed in order to overcome the problem of uncontrolled priority inversion. For example, if transaction T blocks higher transactions, T inherits $P_H$, the highest priority of the transactions blocked by T.

Under this protocol, it is not necessary to check for the possibility of read-write conflicts. For instance, when a data object is write-locked by a transaction, the rw-priority ceiling is equal to the highest priority transaction that can access it. Hence, the protocol will block a higher priority transaction that may write or read it. On the other hand, when the data object is read-locked, the rw-priority ceiling is equal to the highest priority transaction that may write it. Hence, a transaction that attempts to write it will have a priority no higher than the rw-priority ceiling and will be blocked. Only the transaction that read it and have priority higher than the rw-priority ceiling will be allowed to read-lock it, since read-locks are compatible. Using the priority ceiling protocol, mutual deadlock of transactions cannot occur and each transaction can be blocked by at most by one lower priority transactions until it completes or suspends itself. The next example shows how transactions are scheduled under the priority ceiling protocol.

*Example:* Consider the same situation as in the previous example. According to the protocol, the priority ceiling of $O_i$ is the priority of $T_1$. When $T_2$ tries to access a data object, it is blocked because its priority is not higher than the priority ceiling of $O_i$. Therefore $T_1$ will be blocked only once by $T_3$ to access $O_i$, regardless of the number of data objects it may access.

The total priority ordering of active transactions leads to some interesting behavior. As shown in the example above, the priority ceiling protocol may forbid a transaction from locking an unlocked data object. At first sight, this seems to introduce unnecessary blocking. However, this can be considered as the "insurance premium" for preventing deadlock and achieving block-at-most-once property.

Using the prototyping environment, we have investigated issues associated with this idea of total ordering in priority-based scheduling protocols. One of the critical issues related to the total ordering approach is its performance compared with other design alternatives. In other words, it is important to figure out what is the actual cost for the "insurance premium" of the total priority ordering approach.

## 4.4. Performance Evaluation

Various statistics have been collected for comparing the performance of the priority-ceiling protocol with other synchronization control algorithms. Transaction are generated with exponentially distributed interarrival times, and the data objects updated by a transaction are chosen uniformly from the database. A transaction has an execution profile which alternates data access requests with equal computation requests, and some processing requirement for termination (either commit or abort). Thus the total processing time of a transaction is directly related to the number of data objects accessed. Due to space considerations, we do not present all our results but have selected the graphs which best illustrate the difference and performance of the algorithms. For example, we have omitted the results of an experiment that varied the size of the database, and thus the number of conflicts, because they only confirm and not increase the knowledge yielded by other experiments.

For each experiment and for each algorithm tested, we collected performance statistics and averaged over the 10 runs. The percentage of deadline-missing transactions is calculated with the following equation: *%missed* = 100 * (number of deadline-missing transactions / number of transactions processed). A transaction is processed if either it executes completely or it is aborted. We assume that all the transactions are *hard* in the sense that there will be no value for completing the transaction after its deadline. Transactions that miss the deadline are aborted, and disappeared from the system immediately with some abort cost. We have used the transaction size (the number of data objects a transaction needs to access) as one of the key variables in the experiments. It varies from a small fraction up to a relatively large portion (10%) of the database so that conflict would occur frequently. The high conflict rate allows synchronization protocols to play a significant role in the system performance. We choose the arrival rate so that protocols are tested in a heavily loaded rather than lightly loaded system. It is because for designing real-time systems, one must consider high load situations. Even though they may not arise frequently, one would like to have a system that misses as few deadlines as possible when such peaks occur. In other words, when a crisis occurs and the database system is under pressure is precisely when making a few extra deadlines could be most important [Abb88].

We normalize the transaction throughput in records accessed per second for successful transactions, not in transactions per second, in order to account for the fact that bigger transactions need more database processing. The normalization rate is obtained by multiplying the transaction completion rate (transactions/second) by the transaction size (database records accessed/transaction).

In Figure 5, the throughput of the priority-ceiling protocol (C), the two-phase locking protocol with priority mode (P), and the two-phase locking protocol without priority mode (L), is shown for transactions of different sizes with balanced workload and I/O bound workload. The two important factors affecting the performance of locking protocols are their abilities to resolve the locking conflicts and to perform I/O and transactions in parallel. When the transaction size is small, there is little locking conflict and the problem such as deadlock and priority inversion has little effect on the overall performance of a locking protocol. On the other hand, when transaction size becomes large, the probability of locking conflicts rises rapidly. In fact, the probability of deadlocks goes up with the fourth power of the transaction size [Gray81]. Hence, we would expect that the performance of protocols will be dominated by their abilities to handle locking conflicts when transaction size is large.

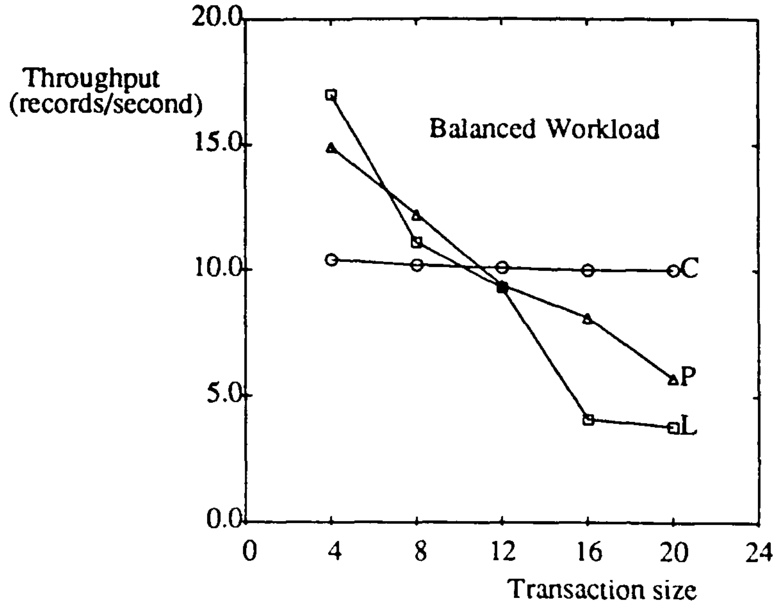
As illustrated in Figure 5, the performance of the two-phase locking protocol, with or without priority assignments to transactions, degrades very fast when transaction size increases. This can be attributed to the inability of this protocol to prevent deadlock and priority inversions. On the other hand, the priority ceiling protocol handles locking conflicts very well. The protocol performs much better than the two-phase locking protocol when the transaction size is large. The main weakness of the priority ceiling protocol is its inability to perform I/O and transactions in parallel. For example, suppose that transaction T has lock on $O_1$ and it now wants to lock data object $O_2$. Unfortunately, $O_2$ is not in the main memory. As a result, T is suspended. However, neither are transactions with priorities lower than the rw-priority ceiling of $O_1$ allowed to execute. This could lead to the idling of the processor until either $O_2$ is transferred to the main memory or a transaction whose priority is higher than the rw-priority ceiling arrives. We refer this type of blocking as I/O blocking. When transaction size is small, the locking conflict rate is small. Hence, the two-phase locking protocol performs well. However, due to I/O blocking the throughput of the priority ceiling protocol is not as good as that of the two-phase locking protocol, especially when the workload is I/O bounded.

Since I/O cost is one of the key parameters in determining performance, we have investigated an approach to improve system performance by performing I/O operation before locking. This is called the *intention I/O*. In the intention mode of I/O operation, the system pre-fetches data objects that are in the access lists of transactions submitted, without locking them. This approach will reduce the locking time of data objects, resulting in higher throughput. As shown in Figure 6, intention I/O improves throughput of both the two-phase locking and the ceiling protocol. However, improvement in the ceiling protocol is much more significant. This is because intention I/O effectively solves the I/O blocking problem of the priority ceiling protocol.

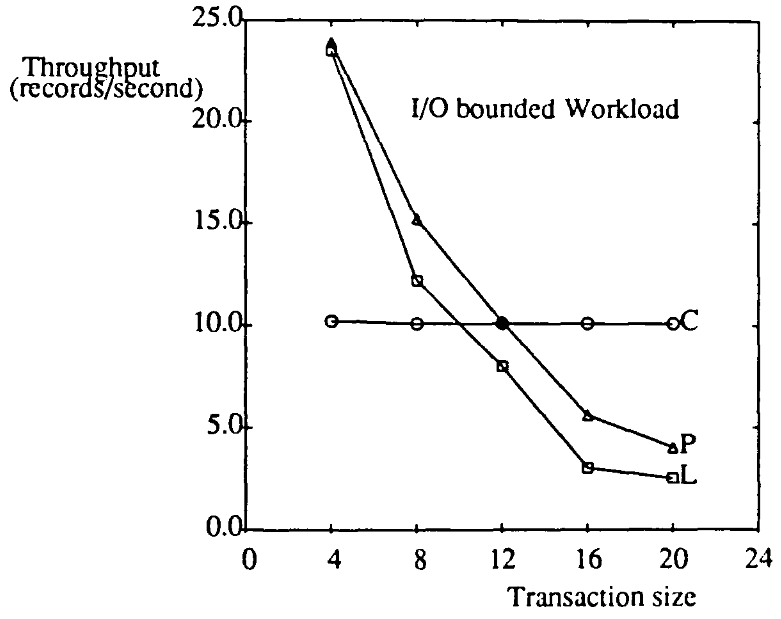
Another important performance statistics is the percentage of deadline missing transactions, since the synchronization protocol in real-time database systems must satisfy the timing constraint of individual transaction. In our experiments, each transaction's deadline is set to proportional to its size and system workload (number of transactions), and the transaction with the earliest deadline is assigned the highest priority. As shown in Figure 7, the percentage of deadline missing transactions increases sharply for the two-phase locking protocol as the transaction size increases due to its inability to deal with deadlock and to give preference to transactions with shorter deadlines. Two-phase locking with priority assignment performs somewhat better, because the timing constraints of transactions are considered, although the deadlock and priority inversion problems still handicap its performance. The priority ceiling protocol has the best relative performance because it addresses both the deadlock and priority inversion problem.

A drawback of the priority ceiling protocol from the practical viewpoint is that it needs knowledge of all transactions that will be executed in the future. This may be a very strong requirement to satisfy in some applications.

The priority ceiling protocol takes a conservative approach. It is based on two-phase locking and employs only blocking, but not roll-back, to solve conflicts. For conventional database systems, it has



a) balanced workload transaction



b) I/O bounded workload transaction

Fig. 5 Transaction Throughput.

C: priority_ceiling protocol
P: 2-phase locking protocol with priority mode
L: 2-phase locking protocol without priority mode

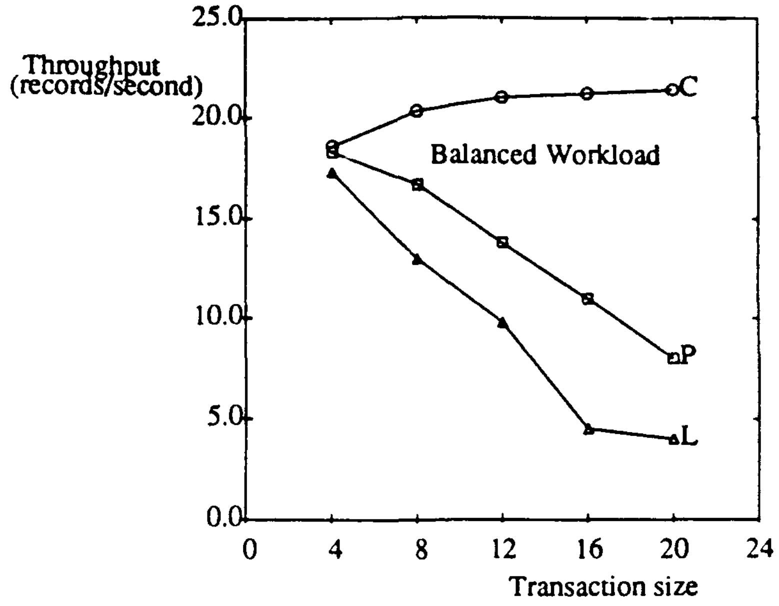


Fig. 6 Transaction Throughput with Intention I/O.

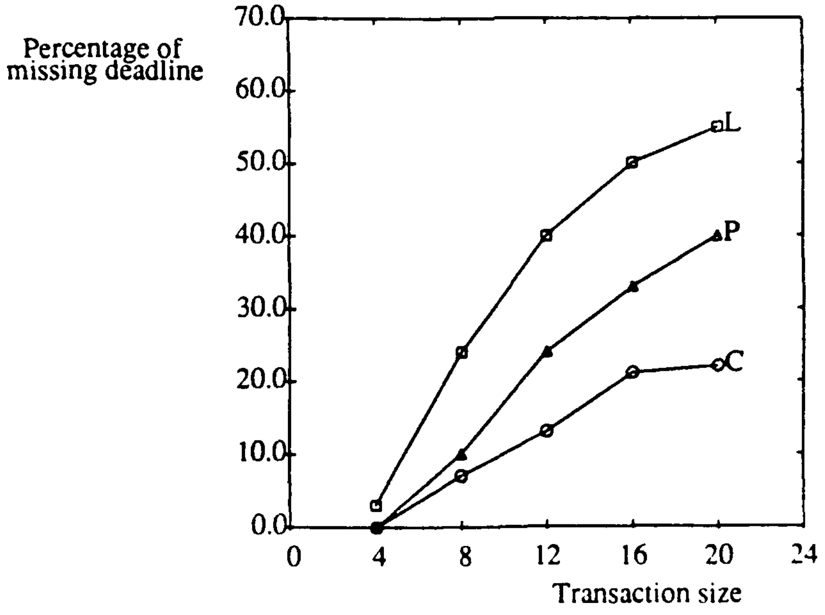


Fig. 7 Percentage of Missing Deadline.

been shown that optimal performance may be achieved by compromising blocking and roll-back [Yu90]. For real-time database systems, we may expect similar results. Aborting a few low priority transactions and restarting them later may allow high priority transactions to meet their deadlines, resulting in improved system performance. Several concurrency control protocols based on optimistic approach have been proposed [Har90, Lin90, Son90]. They incorporate priority-based conflict resolution mechanisms, such as *priority wait*, that makes low priority transactions wait for conflicting high priority transactions to complete. However, this approach of detecting conflicts during validation phase degrades system predictability. A transaction is detected as being late when it actually misses its deadline, since the transaction is only aborted in the validation phase.

## 4.5. A Multiversion Database System

To illustrate the effctiveness of the prototyping environment, we have investigated the performance of a multiversion database system. There is no correlation between the priority ceiling protocol study and multiversion database study.

In a multiversion database system, each data object consists of a number of consecutive versions. The objective of using multiple versions in real-time database systems is to increase the degree of concurrency and to reduce the possibility of rejecting user requests by providing a succession of views of data objects. One of the reasons for rejecting a user request is that its operations cannot be serviced by the system. For example, a read operation has to be rejected if the value of data object it was supposed to read has already been overwritten by some other user request. Such rejections can be avoided by keeping old versions of each data object so that an appropriate old value can be given to a tardy read operation. In a system with multiple versions of data, each write operation on a data object produces a new version instead of overwriting it. Hence, for each read operation, the system selects an appropriate version to read, enjoying the flexibility in controlling the order of read and write operations. When a new version is created, it is *uncertified*. Uncertified versions are prohibited from being read by other transactions to guarantee cascaded-abort free [Bern87]. A version is *certified* at the commit time of the transaction that generated the version.

The multiversion database system we have implemented is based on timestamp ordering. Each data object is represented as a list of versions, and each version is associated with timestamps for its creation and the latest read, and a valid bit to specify whether the version is certified. The multiversion concurrency control scheme we have implemented is called "multiversion timestamp ordering method", and is proved to satisfy the serializability [Bern87].

Each transaction consists of read and write requests for data objects. Read requests are never rejected in a multi-version database system if all the versions are retained. A read operation does not necessarily read the latest committed version of a data object. A read request is transformed to a version-read operation by selecting an appropriate version to read. The timestamp of a read request is compared with the write-timestamp of the highest available version. When a read request with timestamp T is sent to the Resource Manager, the version of a data object with the largest timestamp less than T is selected as the value to be returned. Figure 8 shows an example of a read operation with the timestamp "11".

The timestamp of a write request is compared with the read timestamp of the highest version of the data object. A new version with the timestamp greater than the read-timestamp of the highest certified version is built on the upper level, with the valid bit reset to indicate that the new version is not certified yet. In order to simplify the concurrency control mechanism, we allow only one temporary version for each data object. Inserting a new version in the middle of existing valid versions is not allowed.

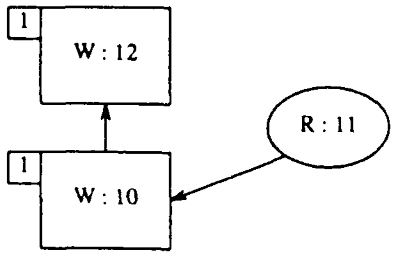


Fig. 8. A read operation with two certified versions of a data object.

The experiment was conducted to measure the average response time and the number of aborts for a group of transactions running on a multiversion database system and its corresponding single-version system. Two groups of transactions with different characteristics (e.g., type and number of access to data objects) were executed concurrently. The objective was to study the sensitivity of key parameters on those two performance measures. Here we present our findings briefly.

Performance is highly dependent on the set size of transactions. As shown in Figure 9, a multiversion database system outperforms the corresponding single-version system for the type of workload under which they are expected to be beneficial: a mix of small update transactions and larger read-only transactions. The reason for this is that, in a multiversion database system, a read requests have higher priority than the write requests; whereas the priority for read requests is not provided in a single-version system. Therefore, in a single-version system, the probability of rejecting a read request is equal to that of a write request. The experiment shows that a single-version database system outperforms its multiversion counterpart for a different transaction mix.

It was observed that the performance of a multiversion system in terms of the number of aborts is better than its single-version counterpart for a mix of small update transactions and larger read-only transactions. Similar experiments have been performed by changing the database size and the mean interarrival time of transactions. It was found, however, that the main result remains the same. From these experiments, it becomes clear that among the four variables we studied, the set-size of transactions is the most sensitive parameter for determining the performance of a multiversion database system. This experiment demonstrates the expressive power and performance evaluation capability of the prototyping environment.

## 5. Conclusions

Prototyping large software systems is not a new approach. However, methodologies for developing a prototyping environment for real-time database systems have not been investigated in depth in spite of its potential benefits. In this paper, we have presented a prototyping environment that has been developed based on the StarLite concurrent programming kernel and message-based approach with modular building blocks. Although the complexity of a distributed database system makes prototyping difficult, the implementation has proven satisfactory for experimentation of design choices, different database control techniques, and even an integrated evaluation of database systems.

There are three main goals to be achieved in developing a prototyping environment for real-time database systems: modularity, flexibility, and extensibility. Modularity enables the environment to be easily reconfigured, since any subset of the available modules can be combined to produce a new testing environment.

An additional benefit of the "right" modularity is that actual system software can be developed in the prototyping environment and then ported to the target machine. This is enabled by the use of technology-independent interfaces which are general enough to support any target system architecture. In addition to the portability, programs may be run in a "hybrid" mode, that is, not all service calls need be simulated. For example, file system calls in the application program can be intercepted by the interpreter and directed to the existing host file system. Then, as a file system is developed, the file system calls can be directed to it. If the file system is not necessary or is not the focus of the current research, it need not be developed. This feature of the prototyping environment allows the developer to focus on only pertinent design issues.

Flexibility enables the prototyping environment to be applicable over a wide range of configurations and system parameters. One of the keys to achieving this goal is to design interfaces whose operations are independent both of the implementation technology and the context in which they are used. For example, the user-level Send operation sends an array of bytes to an abstract data type, the PortTag. Thus this operation can be used to send any packet type to any destination, be it local or distant.

The third goal is that the prototyping environment be extensible enough to model additional features of particular systems by adding modules without affecting the operation of or requiring the recompilation of existing modules. For instance, the implementation can be extended to model the operation of different types of I/O devices of different speeds by modifying the implementation module that performs the read and write operations. One way to modify the implementation would be to delay for a period depending on the address passed to the read or write operation. Reading from a disk might be indicated by one range of addresses and take some time, while reading from a tape drive might be indicated by another range and presumably take longer. However, because the interface of this module is device-independent, changing the implementation to process I/O requests at different speed will not affect any of the modules that request I/O operations. Therefore, time and effort for system reconfiguration can be reduced.

Expressive power and performance evaluation capability of our prototyping environment has been demonstrated by implementing real-time database systems and investigating the performance characteristics of the priority-ceiling protocol and multiversion databases.

In real-time database systems, transactions must be scheduled to meet their timing constraints. In addition, the system should support a predictable behavior such that the possibility of missing deadlines of critical tasks could be informed ahead of time, before their deadlines expire. Priority ceiling protocol is one approach to achieve a high degree of schedulability and system predictability. In this paper, we have investigated this approach and compared its performance with other techniques and design choices. It is shown that this technique might be appropriate for real-time transaction scheduling since it is very stable over the wide range of transaction sizes, and compared with two-phase locking protocols, it reduces the number of deadline-missing transactions.

Using the prototyping environment, we have shown that in general, a database system with a multiversion concurrency control algorithm performs better while processing read requests. Read requests that would be aborted in a single-version database system due to conflicts may be successfully processed in a multiversion system using older versions. Therefore, when the read requests dominate the transaction load, and there is a high probability for abort of read-only transactions due to conflicts, a multiversion system outperforms its corresponding single-version system. The relative sizes of the read and write sets of transactions is an important factor affecting the performance. Although the actual performance figures will vary depending on workload and implementation details, we believe that our results provide a good picture of the costs and benefits associated with the multiversion approach to concurrency control.

Real-time distributed database systems need further investigation. In priority ceiling protocol and many other database scheduling algorithms, preemption is usually not allowed. To reduce the number of deadline-missing transactions, however, preemption may need to be considered. The preemption decision in a real-time database system must be made very carefully, and as pointed out in [Stan88], it should not necessarily based only on relative deadlines. Since preemption implies not only that the work done by the preempted transaction must be undone, but also that later on, if restarted, must redo the work. The resultant delay and the wasted execution may cause one or both of these transactions, as well as other transaction to miss the deadlines. Several approaches to designing scheduling algorithms for real-time transactions have been proposed [Liu87, Stan88, Abb88], but their performance in distributed environments is not studied. The prototyping environment described in this paper is an appropriate research vehicle for investigating such new techniques and scheduling algorithms for real-time database systems.

## References

[Abb88] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Study," *VLDB Conference*, Sept. 1988, pp 1-12.

[Bern87] Bernstein, P., V. Hadzilacos, and N. Goodman, *Concurrency Control and recovery in Database Systems*, Addison Wesley, 1987.

[Bri78] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *Comm. of the ACM 21*, 11, Nov. 1978.

[Cook87] Cook, R. and S. H. Son, "The StarLite Project," *Fourth IEEE Workshop on Real-Time Operating Systems*, Cambridge, Massachusetts, July 1987, 139-141.

[Gray81] Gray, J. et al., "A Straw Man Analysis of Probability of Waiting and Deadlock," *IBM Research Report*, RJ 3066, 1981.

[Har90] Haritsa, J., M. Carey, and M. Livny, "On Being Optimistic on Real-Time Constraints," *ACM PODS Symposium*, April 1990.

[Hua90] Huang, J., J. Stankovic, D. Towsley, and K. Ramamritham, "Real-Time Transaction Processing: Design, Implementation and Performance Evaluation," *Tech. Rep. TR-90-43*, Dept. of Computer and Information Science, University of Massachusetts, May 1990.

[IEEE90] *Seventh IEEE Workshop on Real-Time Operating Systems and Software*, University of Virginia, Charlottesville, Virginia, May 1990.

[Kiv69] Kiviat, P., R. Villareau, and H. Markowitz, *The SIMSCRIPT II Programming Language*, Englewood Cliffs, NJ, Prentice-Hall, 1969.

[Kor90] Korth, H., "Triggered Real-Time Databases with Consistency Constraints," *16th VLDB Conference*, Brisbane, Australia, Aug. 1990.

[Lam78] L. Lamport, "Time, Clocks and Ordering of Events in Distributed Systems," *Commun. ACM*, July 1978, pp 558-565.

[Lin89] Lin, K., "Consistency issues in real-time database systems," *Proc. 22nd Hawaii Intl. Conf. System Sciences*, Hawaii, Jan. 1989.

[Lin90] Lin, Y. and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *11th IEEE Real-Time Systems Symposium*, Orlando, Florida, Dec. 1990.

[Liu87] Liu, J. W. S., K. J. Lin, and S. Natarajan, "Scheduling Real-Time, Periodic Jobs Using Imprecise Results," *Real-Time Systems Symposium*, Dec. 1987, 252-260.

[ONR90] *ONR Workshop on Foundations of Real-Time Computing*, Washington, D. C., Oct. 1990.

[Raj89] Rajkumar, R., "Task Synchronization in Real-Time Systems," *Ph.D. Dissertation*, Carnegie-Mellon University, August 1989.

[Sch74] Schriber, T., *Simulation Using GPSS*, NY, Wiley, 1974.

[Sha87] Sha, L., R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocol: An Approach to Real-Time Synchronization," *Technical Report*, Computer Science Dept., Carnegie-Mellon University, 1987.

[Sha88] Sha, L., R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988, 82-98.

[Shin87] Shin, K. G., Introduction to the Special Issue on Real-Time Systems, *IEEE Trans. on Computers*, Aug. 1987, 901-902.

[Son88] Son, S. H., editor, *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988.

[Son88b] Son, S. H., "A Message-Based Approach to Distributed Database Prototyping," *Fifth IEEE Workshop on Real-Time Software and Operating Systems* Washington, DC, May 1988, 71-74.

[Son89] Son, S. H. and A. Agrawala, "Distributed Checkpointing for Globally Consistent States of Databases," *IEEE Transactions on Software Engineering*, Vol. 15, No. 10, October 1989, 1157-1167.

[Son89b] Son, S. H. and H. Kang, "Approaches to Design of Real-Time Database Systems," *Symposium on Database Systems for Advanced Applications*, Korea, April 1989, pp 274-281.

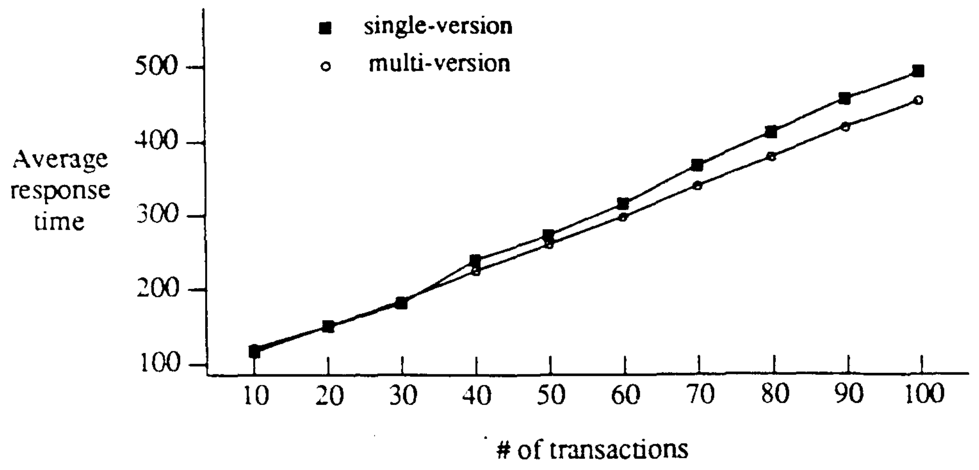
[Son90] Son, S. H. and J. Lee, "Scheduling Real-Time Transactions in Distributed Database Systems," *7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, Virginia, May 1990, pp 39-43.

[Stan88] Stankovic, J., "Misconceptions about Real-Time Computing," *IEEE Computer 21*, 10, October 1988, pp 10-19.

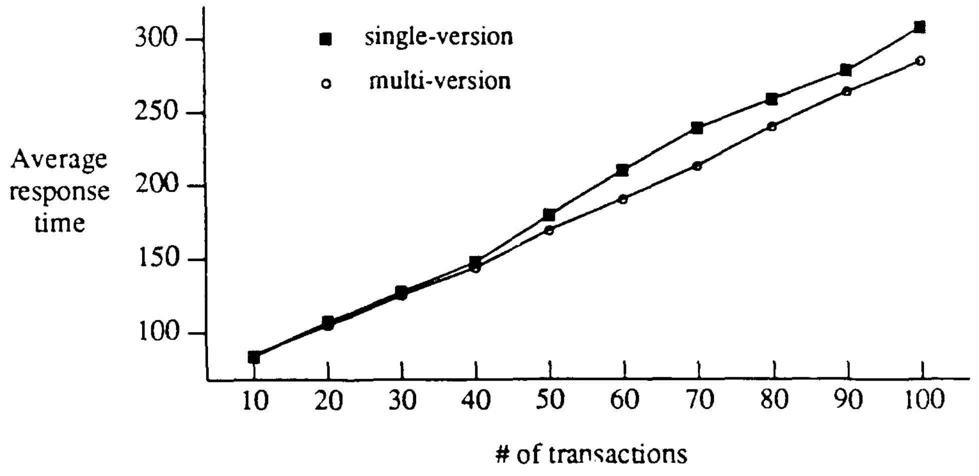
[Tan87] Tanenbaum, A., *Operating Systems Design and Implementation*, Prentice-Hall, 1987.

[Tok89] Tokuda, H. and C. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, 23 (3), July 1989.

[Yu90] Yu, P. and D. Dias, "Concurrency Control using Locking with Deferred Blocking," *6th Intl. Conf. Data Engineering.*, Los Angeles, Feb. 1990, pp 30-36.



**PARAMETERS**
Group 1 : Setsize = 10, Type = READ-only, Transaction Ratio = 80%
Group 2 : Setsize = 2, Type = WRITE-only, Transaction Ratio = 20%



**PARAMETERS**
Group 1 : Setsize = 10, Type = READ-only, Transaction Ratio = 50%
Group 2 : Setsize = 2, Type = WRITE-only, Transaction Ratio = 50%

Fig. 9. Average transaction response time

## DISTRIBUTION LIST

|  |  |
|---|---|
| 1 - 3 | Scientific Officer Code: 1211<br>Dr. James G. Smith<br>Office of Naval Research<br>800 North Quincy Street<br>Arlington, VA 22217-5000 |
| 4 | Administrative Grants Officer<br>Office of Naval Research<br>Resident Representative N66002<br>Administrative Contracting Officer<br>National Academy of Sciences<br>2135 Wisconsin Avenue, N. W., Suite 102<br>Washington, DC 20007-3259 |
| 5 | Director, Naval Research Laboratory<br>Attn: Code 2627<br>Washington, DC 20375 |
| 6 - 7 | Defense Technical Information Center<br>Building 5, Cameron Station<br>Alexandria, Virginia 22314 |
| 8 - 9 | S. H. Son |
| 10 | A. K. Jones |
| 11 - 12 | E. H. Pancake, Clark Hall |
| 13 | SEAS Preaward Administration Files |

JO#4043:ph